

# Minimal Static Single Assignment Form

Masterarbeit von

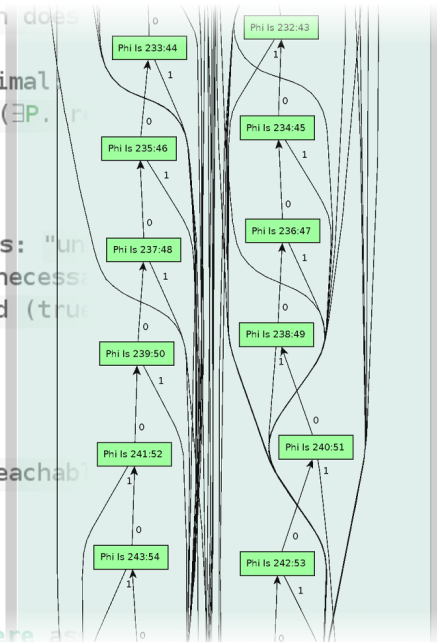
**Maximilian Wagner**

an der Fakultät für Informatik

```

txt <Theorem 1. A graph which does
theorem no_redundant_set_minimal
assumes no_redundant_set: "¬(∃p. r
shows "cytronMinimal g"
proof (rule ccontr)
  assume "¬cytronMinimal g"
  then obtain φ where φ_props: "un
  using cytronMinimal_def unnecess
  consider (nontrivial) "card (tru
  thus False
proof cases
  case trivial
  from this φ_props(1)
  have "redundant_set g (reachab
  with no_redundant_set
  show False by simp
next
  case nontrivial
  then obtain r s φr φs where a

```



**Erstgutachter:** Prof. Dr.-Ing. Gregor Snelting  
**Zweitgutachter:** Prof. Dr. rer. nat. Bernhard Beckert  
**Betreuende Mitarbeiter:** Dipl.-Inform. Denis Lohner  
 Dipl.-Inform. Sebastian Buchwald

**Bearbeitungszeit:** May 25, 2016 – November 24, 2016



# Zusammenfassung

Die meisten modernen Compiler benutzen Static Single Assignment (SSA) Form als Zwischenrepräsentation (*intermediate representation*, IR). Daher sind effiziente SSA-Aufbau-Algorithmen mit möglichst kompakter Ausgabe essentiell für die Performance von Compilern und für die Qualität des generierten Codes.

Braun et al. [1] veröffentlichten kürzlich einen einfachen SSA-Aufbau-Algorithmus, dessen Laufzeit sich ähnlich verhält wie die des Algorithmus von Cytron et al. [2], welcher sich als Standard etabliert hat. In seiner einfachen Form garantiert der Algorithmus von Braun et al. jedoch nur dann minimale IR-Größe, wenn das Eingabeprogramm reduzierbaren Kontrollfluss aufweist. Braun et al. geben daher eine Erweiterung ihres Algorithmus' an, welche auch für irreduzierbaren Kontrollfluss ein minimales Ergebnis liefert.

In dieser Masterarbeit beweisen wir mittels des Theorembeweislers Isabelle/HOL [3], dass diese Erweiterung korrekt ist. Außerdem zeigen wir, dass die von der Minimierungserweiterung hergestellte Grapheigenschaft genügt, um SSA-Minimalität nach Cytron et al. zu folgern. Wir liefern eine Analyse der Laufzeitkomplexität dieser Erweiterung in Abhängigkeit der  $\phi$ -Funktionen, die zum Minimierungszeitpunkt vorhanden sind.

Des Weiteren evaluieren wir die Laufzeit und die Notwendigkeit von SSA-Minimierung anhand einer C-Implementierung dieser Erweiterung. Wir stellen fest, dass der SSA-Aufbau nach Braun et al. in Realwelt-Situationen selten überschüssige  $\phi$ -Funktionen generiert. Die zusätzliche Dauer der SSA-Minimierung ist in unseren Experimenten zwar vernachlässigbar gering, jedoch bietet die SSA-Minimierung in den meisten Fällen keine signifikante Verbesserung in der Compiler-Ausgabe. Wir schließen daraus, dass die Situationen, in denen dieser Minimierungsschritt die Performance des generierten Codes messbar verbessert hat, sehr selten sind. Diese Instanzen finden sich ironischerweise nur bei Code, der manuell für Performance optimiert wurde. Nichtsdestotrotz bleibt der SSA-Aufbau-Algorithmus von Braun et al. sogar ohne den Minimierungsschritt ein guter Kompromiss zwischen Implementierungskomplexität, Größe der resultierenden SSA-Form und Laufzeit.

# Abstract

Most modern compilers use Static Single Assignment (SSA) form as intermediate representation. Having an efficient algorithm that outputs the most concise SSA representation of the input program is thus important for compiler performance and the quality of the generated code. Recently, Braun et al. [1] have presented a simple SSA construction algorithm that achieves similar performance to the previous de facto standard algorithm by Cytron et al. [2]. However, given an input program with irreducible control flow, the simple version of Braun et al.'s algorithm generates suboptimal output in terms of the size of the generated SSA form. Braun et al. also provide an extension to their algorithm which ensures minimal output even for irreducible control flow.

This thesis concerns itself exclusively with that extension. Using the theorem prover Isabelle/HOL [3], we formally prove the correctness of this extension. We then prove that the property established by this algorithm is at least as good a criterion for minimality as the minimality guaranteed by Cytron et al.'s algorithm. We give a complexity analysis in terms of the relevant part of the SSA representation and further evaluate the performance of the algorithm via an implementation in C. We observe that Braun et al.'s algorithm rarely produces suboptimal SSA form given real-world programs as input. In our experiments, the additional minimization step only took an insignificant amount of time during compilation, but the improvement to the performance of compiler output was negligible in most cases. We conclude that while the cases in which minimization measurably benefits the performance of generated code are rare, they tend to coincide with code which was, ironically, hand-tuned for performance. Nevertheless, Braun et al.'s SSA construction algorithm remains a good choice of algorithm, even without minimization.

# Erklärung

Hiermit erkläre ich, Maximilian Wagner, dass ich die vorliegende Masterarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

---

Ort, Datum

---

Unterschrift



# Contents

<b>1. Introduction</b>	<b>9</b>
<b>2. Background</b>	<b>11</b>
2.1. (Ir-)Reducible Control Flow . . . . .	11
2.2. SSA Form . . . . .	13
2.2.1. Cytron-Minimality . . . . .	14
2.2.2. Trivial, Unnecessary and Redundant $\phi$ -Functions . . . . .	15
2.3. Isabelle . . . . .	15
<b>3. Formalization and Proof</b>	<b>17</b>
3.1. Framework . . . . .	17
3.2. Redundant Sets Contain Redundant SCCs . . . . .	19
3.3. Non-redundancy Implies Cytron-Minimality . . . . .	21
<b>4. Design and Implementation</b>	<b>27</b>
4.1. Braun et al.'s Peeling Algorithm . . . . .	27
4.1.1. Description of the Algorithm . . . . .	27
4.1.2. Algorithmic Complexity . . . . .	29
4.2. Implementation . . . . .	32
<b>5. Evaluation</b>	<b>35</b>
5.1. Occurrence of Redundant $\phi$ -Functions . . . . .	35
5.2. Effect of Redundant $\phi$ -Functions on Performance . . . . .	39
5.3. Sources of Redundant $\phi$ -Functions . . . . .	41
<b>6. Related Work</b>	<b>45</b>
<b>7. Conclusion</b>	<b>47</b>
<b>A. Appendices</b>	<b>53</b>
A.1. Full Isabelle Proof Document . . . . .	53
A.2. Implementation of the Peeling Algorithm . . . . .	77
A.3. Code Used for Worst-Case Example . . . . .	83





# 1. Introduction

Most modern compilers use an intermediate representation based on Static Single Assignment Form (SSA form). As such it's important to ensure that the algorithms used for SSA construction produce high-quality SSA form and are efficient. The classical algorithm used for SSA construction is the algorithm by Cytron et al. [2], which constructs *minimal SSA form* with reasonable performance. This algorithm is based on dominance analysis. If the output is to be “pruned”, i.e. free of SSA values with no usage site, it requires an additional liveness analysis. Recently, Braun et al. [1] have presented a simpler algorithm with similar performance that does not require additional analyses for constructing SSA form. The only caveat of this algorithm is that it only guarantees minimality when program control flow is reducible. This property is usually associated with a program being free of GOTOS, however irreducibility can also occur via other means. For instance, certain language constructs (e.g. interleaved switch-case statements and loops) and certain optimizations such as jump threading can induce irreducibility. Even in cases in which control flow remains reducible at all time, certain optimizations (e.g. global value numbering) can modify the SSA graph in ways that give rise to non-minimal constructions [1]. The original paper by Braun et al. also provides an extension to the algorithm which (re-)establishes minimality even in such cases, at the cost of having to perform an additional step after SSA construction.

This extension, which we call Braun et al.'s “peeling algorithm” (due to it recursively “peeling off” nodes from strongly connected components (SCCs) to find successively nested SCCs), can be implemented as an independent compiler pass and can be run at any time between SSA construction and destruction. As a standalone algorithm, it establishes the property of being free of sets of  $\phi$ -functions which, collectively, refer only to one outside value. Using the Isabelle graph framework used in Buchwald et al. [4], we formally prove that this algorithm really does establish this property. We then prove that given Conventional SSA form [5], this property implies that the resulting graph is minimal (as defined by Cytron et al. [2]). This in turn means that freedom of such sets of  $\phi$ -functions can be seen as a definition of minimality on SSA graphs. Being independent of the algorithm used in SSA construction and whether the graph is in conventional SSA form, this constitutes a more general criterion for minimality than that used by Cytron et al. The fact that this type of minimality (together with prunedness and CFG reducibility) guarantees the minimal number of  $\phi$ -functions in any SSA translation of a program [6] gives this additional weight.

---

We also turn our attention to the practical considerations of the peeling algorithm. We perform an analysis of the asymptotic runtime complexity, proving a worst-case bound of  $\mathcal{O}(|V|^2 + |V||E|)$  (in terms of the graph induced by  $\phi$ -functions alone) and present a concrete example of worst-case instances for the algorithm. We've written an implementation of the peeling algorithm in C as a standalone optimization pass in the FIRM compiler back end and provide an analysis of its performance both in the worst case and under common work loads. We conclude from our experiments that the execution of the peeling algorithm does not significantly contribute to compilation time, and that realistic input programs rarely exhibit the kinds of control flow structures that lead to suboptimal SSA form in Braun et al.'s construction algorithm. Braun et al.'s original SSA construction algorithm thus provides an excellent compromise between ease of implementation and size of the output SSA, even without implementing the peeling algorithm.

The rest of this thesis is structured as follows: Section 2 contains an explanation of the prerequisites to understanding this thesis, and an introduction to the syntax used by Isabelle. Section 3 presents the formalization of SSA form we use and explains in detail the proofs conducted in the formal part of this thesis. Section 4 explains the details of the algorithm and conducts an asymptotic runtime complexity analysis. This is followed by a description of the C implementation and a preliminary performance analysis given a worst-case graph instance. Section 5 evaluates the peeling algorithm by running it on a variety of benchmark programs and real-world programs. We then attempt to draw conclusions as to the necessity of this algorithm; how often Braun et al.'s algorithm actually produces suboptimal SSA form given real-world input, and how big of a performance deficit this produces in compiler output. Section 6 discusses the recent advancements in the field of theorem prover assisted compiler construction, and gives an overview of the academic context in which this thesis exists. Finally, Section 7 summarizes the results of our work, points to possible future work and the open questions remaining, and concludes our work.

## 2. Background

This section introduces the necessary concepts for understanding the rest of the thesis. We cover the details of control flow reducibility, SSA form, cytron-minimality, the different kinds of  $\phi$ -functions, and the necessary basics for understanding the Isabelle syntax used in this thesis.

### 2.1. (Ir-)Reducible Control Flow

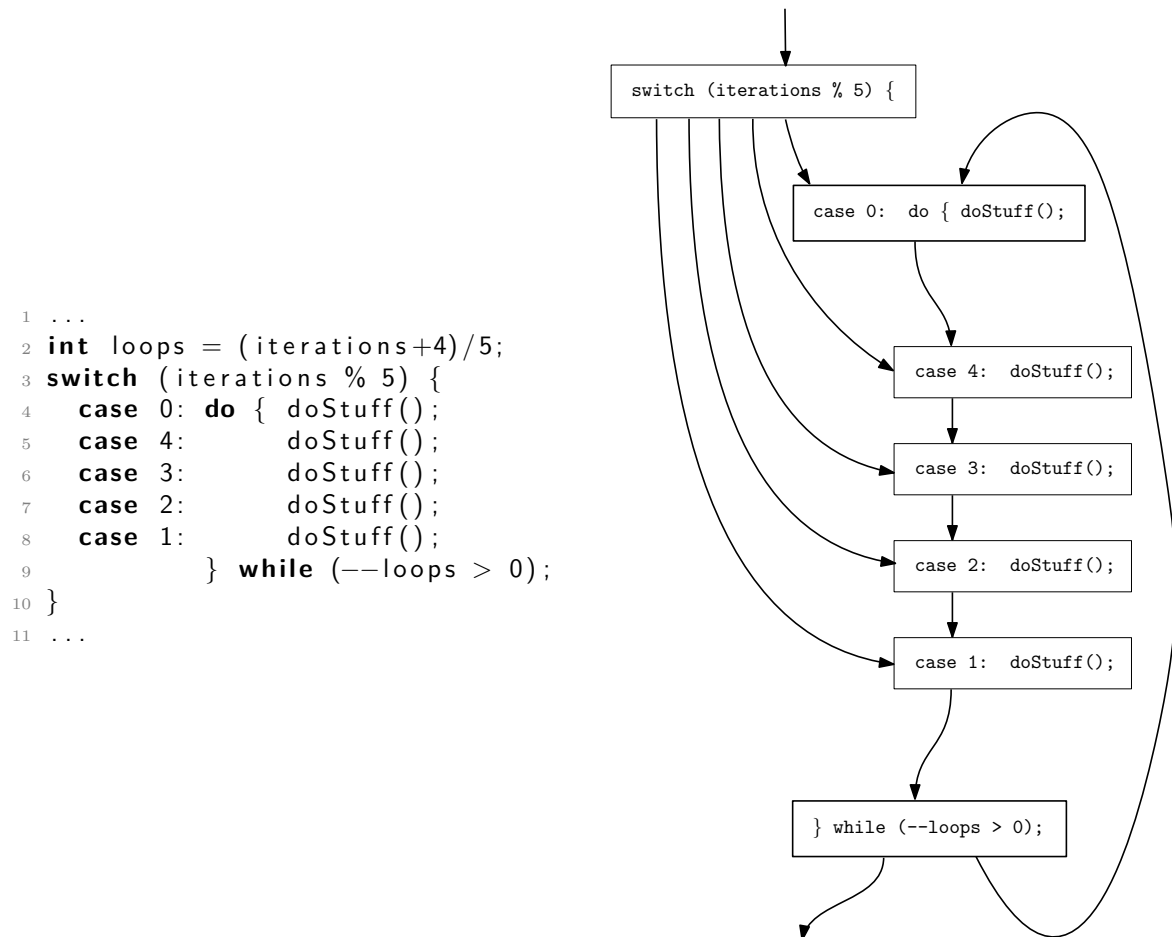
As this work concerns itself with minimization of SSA control flow graphs (CFGs) in the case of irreducible control flow, a discussion of what reducibility is and which mechanisms may violate it is needed. To give a formal definition of reducibility, we first need the notion of dominance.

**Definition.** (*Dominance*) A CFG node  $v$  dominates another node  $w$  iff every path in the CFG from the program's entry point to  $w$  includes  $v$ . We say  $v$  is a *dominator* of  $w$ .

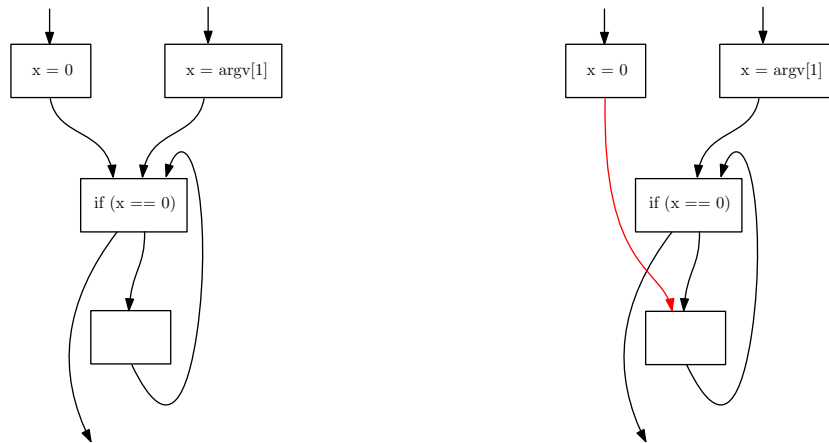
For instance, the program's entry point dominates every node in the CFG. Dominance is reflexive, i.e. every node dominates itself. We can now articulate a formal definition of reducibility [7]:

**Definition.** (*Reducible CFG*) A CFG is reducible iff its edges can be partitioned into two sets  $E_1$  and  $E_2$  such that (the graph induced by)  $E_1$  is acyclic and for every edge  $(v, w) \in E_2$ ,  $w$  dominates  $v$ .

Irreducibility is often associated with the GOTO statement, as this statement allows for the construction of arbitrary control flow. However, any mechanism which may introduce a second entry point into a control flow cycle implies the potential for irreducibility. For instance, a language that allows unstructured interleaving of control flow structures implicitly allows for irreducibility. Figure 2.1 shows the perhaps most famous example of this, Duff's Device.



**Figure 2.1.:** Duff's Device and its representation as a CFG. Note the multiple entry points into the loop.



**Figure 2.2.:** The control flow optimizations made by jump threading may introduce irreducible control flow.

Another way irreducibility might appear during compilation is during optimization. Such optimizations may duplicate, delete, or rewire control flow to improve the performance of the generated code, and in doing so, generate irreducible control flow. Figure 2.2 shows one example of a situation where this arises due to jump threading.

## 2.2. SSA Form

Static single assignment (SSA) form is a property of intermediate representations (IR) requiring that each variable be statically assigned only once. Thus, after transforming a control flow graph (CFG) into SSA form, each *variable* has been transformed into a set of immutable *values*. Situations in which the existence of multiple paths in the CFG leads to multiple definitions reaching a certain point are abstracted over via a special type of value called  $\phi$ -functions. Such  $\phi$ -functions have other SSA values as parameters and formally multiplex (i.e. select) between them at run time depending on the CFG edge taken to reach them. During compilation however, they can be seen as opaque values used for satisfying the requirements of SSA form.

SSA form is used by most modern compilers<sup>1</sup> as its explicit representation of def-use relationships makes data-flow analysis much easier. Additionally, there are a number of IR optimizations that benefit from the CFG being in SSA form, e.g. code motion, common subexpression elimination and constant propagation. Next we want to introduce the conventional property [5]. To do this, we first need to define  $\phi$  webs.

<sup>1</sup>For a non-exhaustive list, see [https://en.wikipedia.org/wiki/Static\\_single\\_assignment\\_form#Compilers\\_using\\_SSA\\_form](https://en.wikipedia.org/wiki/Static_single_assignment_form#Compilers_using_SSA_form).

**Definition.** ( *$\phi$  web*) The set of  $\phi$ -functions which are transitively connected form a  $\phi$  web (or  $\phi$  net).

**Definition.** (*Conventional SSA form (CSSA)*) An SSA CFG is in CSSA form iff all values in a  $\phi$  web and all values directly connected to it via usage edges are interference-free, i.e. a loop-free path from any definition to its usage site does not contain another definition in this set.

Intuitively, this definition coincides with the concept that all SSA values explicitly reference the values that might define their runtime value based only on the structure of the CFG.

### 2.2.1. Cytron-Minimality

The classical algorithm for constructing SSA form [2] implies a criterion for minimality on certain SSA graphs, which we want to reason about. To this end, a certain amount of understanding of Cytron et al.'s algorithm is necessary. We thus introduce the necessary concepts in this section, starting with so-called *iterated dominance frontiers*.

**Definition.** The *dominance frontier*  $DF(n)$  of a node  $n$  is the set of nodes which have a predecessor dominated by  $n$ , but which are not dominated by  $n$  themselves.

In a similar vein, the *iterated dominance frontier* of a node is the union of  $DF(n)$  with all dominance frontiers of nodes in  $DF(n)$ , and so on.

**Definition.** (*Necessary  $\phi$ -function*) A  $\phi$ -function  $f$  for a variable  $V$  is necessary iff  $f$  is contained in a basic block  $Z$  and there are basic blocks  $X$  and  $Y$  containing definitions of  $V$  and nonnull control flow paths  $X \xrightarrow{+} Z$  and  $Y \xrightarrow{+} Z$  such that these paths share nothing but  $Z$ .

**Definition.** (*Cytron-minimality*) A program is in cytron-minimal SSA form if it is in SSA form and if all  $\phi$ -functions are necessary.

Stated more succinctly, a CFG is in cytron-minimal SSA form if all  $\phi$ -functions are placed at the iterated dominance frontiers of definitions.

This definition of minimality still allows for dead  $\phi$ -functions to exist (i.e.  $\phi$ -functions with no usage site or which are transitively only used by other  $\phi$ -functions). The property of being free of dead  $\phi$ -functions is called *prunedness*.

It’s interesting to note that pruned cytron-minimal SSA CFGs are “truly” minimal, in the sense that among all valid translations of the original program, such SSA representations have the smallest number of  $\phi$ -functions (as proven by Ullrich and Lohner [6] in the associated material to the CC paper by Buchwald et al. [4]).

Buchwald et al. [4] pointed out that modifying Cytron et al.’s condition for necessary  $\phi$ -functions to require a corresponding usage site yields a sufficient condition for placing  $\phi$ -functions, even in pruned SSA graphs. This is what we call the *convergence property*, and is defined as follows:

**Definition.** (*Convergence property*) Let  $Z$  be a basic block containing no definition for a variable  $V$ . Let  $Z$  further fulfill the requirements for bearing a necessary  $\phi$ -function for  $V$ . If there is a basic block  $M$  with a usage site for  $V$  and a path  $Z \xrightarrow{\pm} M$  containing no other definitions of  $V$ , then  $Z$  bears a  $\phi$ -function.

### 2.2.2. Trivial, Unnecessary and Redundant $\phi$ -Functions

We call  $\phi$ -functions which are not *necessary* by the above definition *unnecessary*  $\phi$ -functions. Unfortunately this term alone doesn’t allow us to articulate the intuitive reason why a  $\phi$ -function might not be strictly needed. We thus differentiate between unnecessary, *trivial* and *redundant*  $\phi$ -functions:

A  $\phi$ -function which uses at most one other value other than itself can be removed from the CFG without changing program semantics by redirecting all users to instead refer to its argument. We call such  $\phi$ -functions trivial.

If, however, there is a set of  $\phi$ -functions which only use at most one value  $v$  not in this set, the complete set can be removed from the CFG by redirecting all references to any of the  $\phi$ -functions in this set to  $v$ . We call such  $\phi$ -functions redundant. Note that for  $\phi$ -functions, trivial implies redundant, and redundant implies unnecessary, but redundant does not imply trivial.

## 2.3. Isabelle

The main proofs of this thesis are written in and verified by the interactive theorem prover Isabelle and more specifically, Isabelle/HOL, Isabelle’s default object logic [3]. In presenting these proofs, we make extensive use of syntax specific to Isabelle and Isabelle/HOL. Hence, in this section, we introduce a minimum of syntax necessary for understanding these excerpts.

Logical implication is denoted by  $\implies$ , while  $\llbracket A; B \rrbracket$  denotes the conjunction of logical statements  $A$  and  $B$ . Isabelle allows a user to open contexts with a given set of assumptions and assumed types. This mechanism, called *locales*, is used extensively. A locale's type parameters are written with a leading apostrophe (e.g. *'node* is the node type in the graph framework we use). Function types are written as *'argumentType*  $\Rightarrow$  *'resultType*. The type of a set of elements of type *'a* is designated by *'a set*. Similarly, a list of elements of type *'a* is denoted by *'a list*. Tuple types use the mathematical notation (*'a*  $\times$  *'b*), with the functions *fst* and *snd* being used for extracting the first and second elements from a tuple, respectively. Relations are represented as sets of tuples; their transitive respectively reflexive-transitive closures are expressed using the postfix operators  $^+$  and  $^*$ . The empty list is written as  $[]$ , list concatenation is done using the  $@$  operator. Additionally, lists can be pulled apart into the first element and the rest (or their head and tail, to use common terminology) using the functions *hd* and *tail*. The functions *last* and *butlast* provide access to the last and second-to-last element respectively. The *map* function applies a given function to every element of a given list and returns the resulting list. The backtick infix operator  $\`$  is similar, but operates on sets.



## 3. Formalization and Proof

This section reviews the formal framework used to formalize the notions of graphs, CFGs, SSA form, etc. necessary for formal verification using Isabelle. It then goes on to present a proof that the property established by Braun et al.'s algorithm extension suffices for minimality.

Definitions are kept in Isabelle syntax, though they are edited for clarity (e.g. locale definitions are shortened and coalesced). The unabridged version of the formalization and proof can be found in Appendix A.1 and in the archive of formal proofs [8].

### 3.1. Framework

This work builds upon the formal Isabelle framework from Ullrich and Lohner [6]. This framework, ultimately based on an abstract graph framework by Nordhoff and Lammich [9], uses Isabelle locales to formalize the notions of CFGs and the properties required by SSA form.

All functions operate on a graph of type  $'g$  with node type  $'node$ .  $\alpha n$  and  $\alpha e$  provide a list of nodes and the edge set, respectively. The locale further demands that these functions interact correctly, e.g. that the edge endpoints are all nodes in  $\alpha n g$ , or that the list of graph nodes consists of distinct entries.

```
locale graph =  
fixes  $\alpha e :: 'g \Rightarrow ('node \times 'node) \text{ set}$   
  and  $\alpha n :: 'g \Rightarrow 'node \text{ list}$   
  and  $inEdges :: 'g \Rightarrow 'node \Rightarrow 'node \text{ list}$   
assumes  $\alpha n\text{-correct}$ :  
   $\alpha n g \supseteq \text{fst } ' \alpha e g \cup \text{snd } ' \alpha e g$   
assumes  $\alpha n\text{-distinct}$ :  
   $\text{distinct } (\alpha n g)$   
assumes  $inEdges\text{-correct}$ :  
   $\text{set } (inEdges g n) = \{f. (f,n) \in \alpha e g\}$   
begin  
  definition  $predecessors g n \equiv \text{map fst } (inEdges g n)$   
end
```

This graph locale is the basis for the locale specifying a well-formed CFG: Here, *Entry* is a designated node functioning as root, i.e. there exists a path from *Entry* to every node. A well-formed CFG is required to follow the definite-assignment rule, i.e. every path from the entry node to a reference to a variable includes an assignment to that variable. The locale further postulates the existence of a path predicate  $g \vdash n - ns \rightarrow m$  which holds iff there is a path *ns* in the CFG *g* from *n* to *m* (with *ns* including the endpoints of the path).

**locale** *CFG-wf* = *graph* +  
**fixes**

*Entry* :: 'g  $\Rightarrow$  'node **and**  
*defs* :: 'g  $\Rightarrow$  'node  $\Rightarrow$  'var set **and**  
*uses* :: 'g  $\Rightarrow$  'node  $\Rightarrow$  'var set

**assumes** *Entry-in-graph*: *Entry*  $g \in \text{set } (\alpha n \ g)$

**assumes** *Entry-unreachable*: *inEdges*  $g \ (\text{Entry } g) = []$

**assumes** *Entry-reaches*:

$n \in \text{set } (\alpha n \ g) \Longrightarrow \exists ns. g \vdash \text{Entry } g - ns \rightarrow n$

**assumes** *def-ass-uses*:

$g \vdash \text{Entry } g - ns \rightarrow m \Longrightarrow \forall v \in \text{uses } g \ m. \exists n \in \text{set } ns. v \in \text{defs } g \ n$

**assumes** *defs-uses-disjoint*:  $n \in \text{set } (\alpha n \ g) \Longrightarrow \text{defs } g \ n \cap \text{uses } g \ n = \{\}$

**assumes** *defs-finite*: *finite* (*defs*  $g \ n$ )

**assumes** *uses-in- $\alpha n$* :  $v \in \text{uses } g \ n \Longrightarrow n \in \text{set } (\alpha n \ g)$

**assumes** *uses-finite*: *finite* (*uses*  $g \ n$ )

The final locale introduces the concepts necessary to work with well-formed SSA CFGs. The  $\phi$ -functions in a graph are represented via a partial function *phis* mapping tuples of (*node*, *ssa-value*) to the  $\phi$ -function's list of arguments. The terms *allDefs* and *allUses* denote the set of variables defined (resp. referred to) in a CFG node by regular assignments or  $\phi$  definitions. Note that our notion of a well-formed SSA CFG includes the *conventional SSA property*. This ensures there are no interfering definitions on a path between a  $\phi$ -function and its arguments. Furthermore, the *CFG-SSA-wf* locale assumes, in the form of the *allDefs-var-disjoint* property, that if two SSA values reside in the same CFG node, they must be associated to different variables in the original program. When analyzing the structure of the SSA CFG for a fixed variable, this property allows us to not bother distinguishing between SSA values and the CFG nodes that harbour them.

**locale** *CFG-SSA-wf* = *CFG-wf* +  
**fixes**

*phis* :: 'g  $\Rightarrow$  'node  $\times$  'val  $\rightarrow$  'val list

**assumes** *phis-finite*: *finite* (*dom* (*phis*  $g$ ))

**assumes** *phis-in- $\alpha n$* : *phis*  $g \ (n, v) = \text{Some } vs \Longrightarrow n \in \text{set } (\alpha n \ g)$

**assumes** *phis-wf*:

$\text{phis } g \ (n, v) = \text{Some } args \Longrightarrow \text{length } (\text{predecessors } g \ n) = \text{length } args$

**assumes** *simpleDefs-phiDefs-disjoint*:

$n \in \text{set } (\alpha n \ g) \Longrightarrow \text{defs } g \ n \cap \text{phiDefs } g \ n = \{\}$

**assumes** *allDefs-disjoint*:

$$\llbracket n \in \text{set } (\alpha n \ g); m \in \text{set } (\alpha n \ g); n \neq m \rrbracket \implies \text{allDefs } g \ n \cap \text{allDefs } g \ m = \{\}$$

**assumes** *allUses-def-ass*:

$$g \vdash \text{Entry } g \text{--} ns \rightarrow m \implies \forall v \in \text{allUses } g \ n. \exists n \in \text{set } ns. v \in \text{allDefs } g \ n$$

**assumes** *Entry-no-phis*:  $\text{phis } g \ (\text{Entry } g, v) = \text{None}$

**assumes** *conventional*:

$$\llbracket g \vdash n \text{--} ns \rightarrow m; n \notin \text{set } (tl \ ns); v \in \text{allDefs } g \ n; v \in \text{allUses } g \ m; x \in \text{set } (tl \ ns); v' \in \text{allDefs } g \ x \rrbracket \implies \text{var } g \ v' \neq \text{var } g \ v$$

**assumes** *phis-same-var*:  $\text{phis } g \ (n, v) = \text{Some } vs \implies v' \in \text{set } vs \implies \text{var } g \ v' = \text{var } g \ v$

**assumes** *allDefs-var-disjoint*:

$$\llbracket n \in \text{set } (\alpha n \ g); v \in \text{allDefs } g \ n; v' \in \text{allDefs } g \ n; v \neq v' \rrbracket \implies \text{var } g \ v' \neq \text{var } g \ v$$

## 3.2. Redundant Sets Contain Redundant SCCs

Having established a formal framework which allows us to reason about SSA CFGs and their properties, we can move on to the proof proper. For proving that Braun et al.’s algorithm extension ensures minimality, we need to prove the following proposition:

**Proposition.** An SSA CFG which is free of sets of redundant  $\phi$ -functions forming strongly connected components (SCC) in an induced  $\phi$  graph, is cytron-minimal.

We do so by first formally verifying the proof of lemma 1 from Braun et al.’s paper, which has the consequence that a graph free of redundant SCCs is free of redundant sets of any kind. We then prove that being free of redundant sets yields cytron-minimality. While the term “strongly connected component” usually means an inclusion-maximal set of nodes with reachability between elements, we use the term exclusively within the context of induced  $\phi$  graphs. For our purposes, the “ $\phi$  graph induced by  $P$ ” shall refer to the graph induced by  $\phi$  argument edges within a set of CFG nodes  $P$  (i.e. a  $\phi$  web restricted to  $P$ ). Formally:

**definition** *induced-phi-graph*  $g \ P \equiv \{(\varphi, \varphi'). \text{ phiArg } g \ \varphi \ \varphi'\} \cap P \times P$

Here, the binary predicate  $\text{phiArg } g \ \varphi \ v$  holds if  $\varphi$  is a  $\phi$ -function which has  $v$  as an argument. We can now formally define the terms “redundant set” and “redundant SCC” needed for a formal verification of lemma 1 as well as the lemma statement itself.

**definition** *redundant-set*  $g \ P \equiv P \neq \{\} \wedge P \subseteq \text{dom } (\text{phi } g) \wedge (\exists v' \in \text{allVars } g. \forall \varphi \in P. \forall \varphi'. \text{ phiArg } g \ \varphi \ \varphi' \longrightarrow \varphi' \in P \cup \{v'\})$

**definition** *redundant-scc*  $g \ P \ \text{scc} \equiv \text{redundant-set } g \ \text{scc} \wedge \text{is-scc } (\text{induced-phi-graph } g \ P) \ \text{scc}$

**lemma 1:**

**assumes** *redundant-set*  $g P$

**shows**  $\exists scc \subseteq P. \text{redundant-scc } g P scc$

The formalization of this lemma follows the proof sketch given by Braun et al. [1] and thus employs condensation graphs, i.e. graphs obtained by contracting all SCCs. We define the edges of this graph by mapping the endpoints of the original edges to their SCCs, and ignoring those edges in the same SCC:

**definition** *condensation-nodes*  $g P \equiv scc\text{-of } (induced\text{-phi-graph } g P) \text{ ` } P$

**definition** *condensation-edges*  $g P \equiv ((\lambda(x,y). (scc\text{-of } (induced\text{-phi-graph } g P) x, scc\text{-of } (induced\text{-phi-graph } g P) y)) \text{ ` } (induced\text{-phi-graph } g P)) - Id$

Being a condensation of a graph, paths in the condensation imply paths in the original graph:

**lemma** *path-in-condensation-impl-path:*

**assumes**  $(a, b) \in (condensation\text{-edges } g P)^+$  **and**  $(\varphi_a \in a)$  **and**  $(\varphi_b \in b)$

**shows**  $(\varphi_a, \varphi_b) \in (induced\text{-phi-graph } g P)^*$

Next, we note that the edge sets of these condensation graphs are finite, since edges are restricted to tuples of  $\phi$ -functions, of which there are only finitely many. Furthermore, condensation graphs are acyclic, as a cycle in the condensation graph would imply a closed path across multiple SCCs, a contradiction against the maximality of SCCs within their graph. Formally:

**lemma** *finite*  $(condensation\text{-edges } g P)$

**lemma** *acyclic*  $(condensation\text{-edges } g P)$

Being finite and acyclic, the condensation graph of a set  $P$  must have a leaf, i.e. a node with no outgoing edges. Such a leaf in the condensation graph corresponds to an SCC  $L$  in  $P$  with no outgoing edges to other nodes of  $P \setminus L$ .

**lemma** *Ex-condensation-leaf:*

**assumes**  $P \neq \{\}$

**shows**  $\exists L. L \in (condensation\text{-nodes } g P) \wedge (\forall scc.(L, scc) \notin condensation\text{-edges } g P)$

With this, we can finally prove lemma 1:

*Proof.* Given a redundant set  $P$ , consider its condensation graph  $P'$  and a leaf  $l$  of  $P'$ . Since  $P$  is a redundant set, all edges from nodes inside  $P$  to nodes outside  $P$  lead to the same SSA value. Because  $l$  is a leaf, nodes in  $l$  have no neighbours in  $P \setminus l$ . This leaves only the single possible neighbour outside of  $P$  as possible neighbor, meaning  $l$  is a redundant SCC.  $\square$

### 3.3. Non-redundancy Implies Cytron-Minimality

Lemma 1 gives us that eliminating all redundant SCCs eliminates all redundant sets. We go on to prove that this suffices for cytron-minimality, independent of whether control flow is reducible. Given that there are no redundant sets in a graph, the assumption that the graph may be non-cytron-minimal leads to a contradiction. However, before we can do this we need to establish additional terminology and prove a certain path extension lemma.

For an unnecessary  $\phi$ -function  $\varphi$ , we define the *reachable-set* of  $\varphi$  as the set of unnecessary  $\phi$ -functions that are connected to  $\varphi$  via a chain of unnecessary  $\phi$ -functions. We further define the *true arguments* of a  $\phi$ -function to be the set of SSA values “just outside” the reachable set, i.e. the first non-unnecessary- $\phi$  SSA values reachable via a chain of  $\phi$  arguments. This definition gives us a simpler way to characterize redundant  $\phi$ -functions: A set of  $\phi$ -functions with less than two true arguments is redundant.

**inductive-set** *reachable* :: 'g  $\Rightarrow$  'val  $\Rightarrow$  'val set

**where** *unnecessaryPhi* g  $\varphi \implies \varphi \in \text{reachable } g \ \varphi$

|  $\llbracket \varphi' \in \text{reachable } g \ \varphi; \text{phiArg } g \ \varphi' \ \varphi''; \text{unnecessaryPhi } g \ \varphi'' \rrbracket \implies \varphi'' \in \text{reachable } g \ \varphi$

**definition** *trueArgs* g  $\varphi \equiv$

$\{\varphi'. \varphi' \notin \text{reachable } g \ \varphi\} \cap \{\varphi'. \exists \varphi'' \in \text{reachable } g \ \varphi. \text{phiArg } g \ \varphi'' \ \varphi'\}$

Note that since  $\phi$  argument edges do not occur between SSA values corresponding to different variables in the original program, all  $\phi$ -functions in the same reachable-set belong to the same variable. We know that there must be a simple path (i.e. a path without repeated nodes) in the CFG from a  $\phi$ -function’s argument to the  $\phi$  itself, and the conventional property guarantees that there are no interfering definitions on this path. We can use this to gradually construct paths from a definition to a  $\phi$ -function impacted by it. However, this alone doesn’t suffice for the path extensions we have to do, as we need the guarantee that this path doesn’t intersect a given second path independent from the first. For this reason we prove the following lemma. The functions *var*, *oldDefs* and *defNode* are defined within the formal framework to obtain the variable to an SSA value, the variable assignments in a node of the original CFG, and the unique CFG node which a given SSA value occupies, respectively.

**lemma** *phiArg-disjoint-paths-extend*:

**assumes** *var* g r = V **and** *var* g s = V

**and**  $V \in \text{oldDefs } g \ m$  **and**  $V \in \text{oldDefs } g \ n$

**and**  $g \vdash m - ms \rightarrow \text{defNode } g \ r$  **and**  $g \vdash n - ns \rightarrow \text{defNode } g \ s$

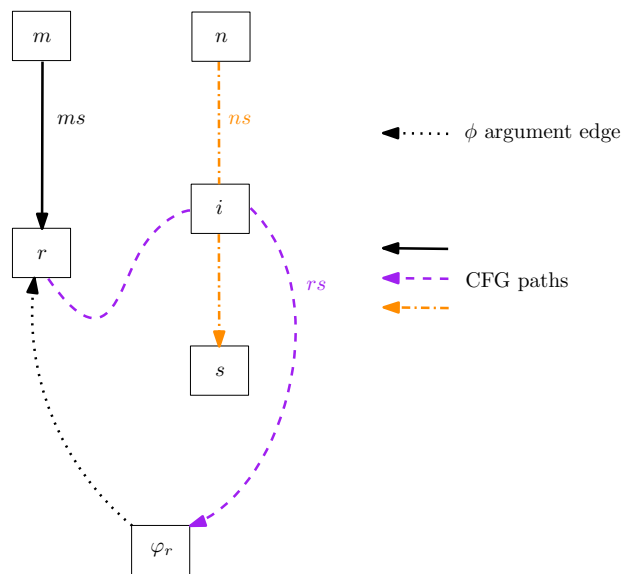
**and**  $\text{set } ms \cap \text{set } ns = \{\}$

**and** *phiArg* g  $\varphi_r \ r$

**obtains**  $ms'$

**where**  $g \vdash m - ms @ ms' \rightarrow \text{defNode } g \ \varphi_r$

**and**  $\text{set } (\text{butlast } (ms @ ms')) \cap \text{set } ns = \{\}$



**Figure 3.1.:** Configuration for the path extension lemma. Note that some paths may be null and some nodes may be aliased.

*Proof.* Assume there were assignments to a variable  $V$  in CFG nodes  $n$  and  $m$  in the original CFG. Assume further that  $r$  and  $s$  are SSA values pertaining to the same variable, and that there are independent paths  $g \vdash m - ms \rightarrow \text{defNode } g \ r$  and  $g \vdash n - ns \rightarrow \text{defNode } g \ s$ . Let  $\varphi_r$  be a  $\phi$ -function with  $r$  as argument.

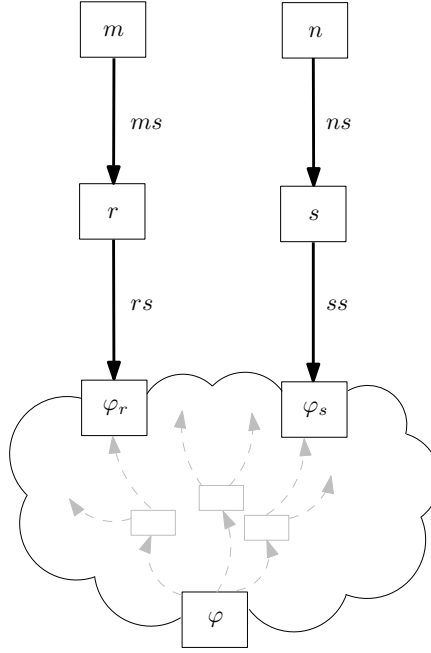
If  $r = \varphi_r$ , the path doesn't need to be extended and the lemma is trivially true. Next, take a simple path from  $r$  to  $\varphi_r$  (guaranteed to exist due to  $r$  being an argument of  $\varphi_r$ ) and call it  $rs$ . If  $rs$  shares no nodes with  $ns$ , then the tail of  $rs$  is the path extension we were looking for. Assume now that  $ns$  and  $rs$  intersect. Then there are three cases that need to be considered for the first point of intersection along  $rs$ , called  $i$  (see Figure 3.1 for an illustration):

- If  $i = \varphi_r$ , then  $\varphi_r$  was on  $ns$  to begin with, and again the tail of  $rs$  suffices as path extension.
- If  $i = n$ ,  $n$  constitutes a definition on the path from a  $\phi$  argument to its  $\phi$ -function, a contradiction to conventional.
- If  $\varphi_r \neq i \neq n$ , then  $i$  is at the convergence point of  $n$  and  $r$ .

In the third case, one might think that  $i$  being the convergence point of two assignments allows us to deduce the existence of a  $\phi$ -function in  $i$ . However,  $\varphi_r$  might be dead, i.e. there might not be a usage site necessitating such a  $\phi$ -function, so this alone does not suffice to deduce the existence of one. Instead, we know by the conventional SSA property that there is no definition pertaining to  $V$  (other than  $r$  and  $\varphi_r$ ) on  $rs$ ,  $rs$  being a path from a  $\phi$  argument to the  $\phi$  itself. Next, remember that the convergence property gives us that “SSA values that are used after convergence points are  $\phi$ -functions and located at the last convergence point”. Indeed, this configuration satisfies the prerequisites for applying the convergence property:

- $i$  is a convergence point of two definitions of  $V$ .
- The SSA value  $r$  is used by  $\varphi_r$ , and there is a path between the two.
- This path  $rs$  is free from definitions for  $V$ .
- $r$  does not correspond to an assignment in the CFG node  $i$ .

Applying the convergence property leaves us with the conclusion that  $r$  must therefore be a  $\phi$ -function located at  $i$ . If the SSA graph is to be well-formed (in that  $r$  may only have one defining CFG node), then  $i$  must have been in  $ms$ , a contradiction to our initial assumption that  $ms$  and  $ns$  share no CFG nodes.  $\square$



**Figure 3.2.:** Configuration in the minimality proof. The cloud represents the reachable-set of  $\varphi$ .

Now that we have all the necessary machinery for the proof proper, we prove it in this form (Given this theorem, the form we need is a simple corollary.):

**theorem** *no-redundant-set-implies-minimality:*  
**assumes**  $\neg(\exists P. \text{redundant-set } g \ P)$   
**shows** *cytronMinimal*  $g$

*Proof.* We employ a proof by contradiction: Assume there are no redundant sets, but that the graph is not cytron-minimal. Then there must be a  $\phi$ -function  $\varphi$  which is not placed at the first convergence point of two variable assignments in the original program. Edge cases notwithstanding, the idea is to take two definitions that impact  $\varphi$  and construct non-crossing paths into its reachable-set, making the  $\phi$  at that location necessary as per Cytron et al.'s definition, a contradiction to the definition of the reachable-set. Figure 3.2 bears an illustration of the CFG configuration constructed in the proof.

Now, because there are no redundant sets,  $\varphi$  has at least two true arguments; were this not the case, its reachable-set would be redundant. There must thus be at least two  $\phi$ -functions  $\varphi_r$  and  $\varphi_s$  in  $\varphi$ 's reachable-set along with distinct non-unnecessary SSA values  $r$  and  $s$  such that  $r$  and  $s$  are arguments of  $\varphi_r$  and  $\varphi_s$ , respectively. Note that  $\varphi_r$  and  $\varphi_s$  might not be distinct. The distinctness of  $r$  and  $s$  implies that there must be distinct definitions  $m$  and  $n$  with  $g \vdash m - ms \rightarrow r$  and  $g \vdash n - ns \rightarrow s$  such that  $\text{set } ns \cap \text{set } ms = \{\}$  holds.



Using our path extension lemma, we can extend these paths to (almost) independent paths leading to  $\varphi_r$  and  $\varphi_s$ . We then have:

$$\begin{aligned} g &\vdash m - ms@rs \rightarrow \varphi_r \\ g &\vdash n - ns@ss \rightarrow \varphi_s \\ \text{set}(\text{butlast}(ns@ss)) \cap \text{set}(\text{butlast}(ms@rs)) &= \{\} \end{aligned}$$

To see why we can find path extensions that fulfill the last property, consider the situation when only one of the two paths has been extended yet (WLOG assume that  $ms$  was extended to  $ms@rs$ ). In that case, we have:

$$\text{set}(\text{butlast}(ms@rs)) \cap \text{set} ns = \{\}$$

However, to perform the second extension, we need to prove that  $\text{set}(ms@rs) \cap \text{set} ns = \{\}$ . The only way that this could be false is if  $\varphi_r$ , being the last element of  $ms@rs$ , is also on the path  $ns$ . If this is the case,  $\varphi_r$  is at the convergence point for the definitions  $m$  and  $n$  (due to the conventional SSA property preventing interference), and is thus a necessary  $\phi$ -function, contradicting with  $\varphi_r \in (\text{reachable } g \varphi)$ . Given this, we can obtain the above statement using a second path extension.

To complete the argument, we note that once we've extended our paths in an intersection-free way into the reachable-set of  $\varphi$ , we know that the convergence point will also lie within this set. This is given to us by the conventional SSA property, which ensures that there are no further definitions on a path from a  $\phi$ -function argument to the corresponding  $\phi$ , combined with the fact that we now have  $\phi$ -argument-chains reaching from  $\varphi$  to outside its reachable-set (and eventually to  $m$  and  $n$  themselves).<sup>1</sup>

As for this convergence point  $\varphi_z$ , it fulfills the definition of being necessary according to Cytron et al., yet we have proven it to lie within the reachable-set of  $\varphi$ , which consists only of unnecessary  $\phi$ -functions, a contradiction.

Thus our initial assumption that a graph can be free of redundant sets without being cytron-minimal must have been false.  $\square$

Given this theorem, the lemma we need is but a simple corollary:

**corollary** *no-redundant-SCCs-implies-minimality:*

**assumes**  $\neg(\exists P \text{ scc. redundant-scc } g P \text{ scc})$

**shows** *cytronMinimal*  $g$

*Proof.* Assume there are no redundant SCCs. Then by lemma 1, there are no redundant sets in  $g$ . Applying the above theorem yields that  $g$  is cytron-minimal.  $\square$

<sup>1</sup>The formal version of this argument is a rather technical nested induction which can be found in the associated material.

Thus an algorithm which eliminates all redundant SCCs in a CSSA CFG establishes cytron-minimality, which is what we set out to prove. Consider that while cytron-minimality implicitly requires the SSA form to be conventional SSA form, we proved lemma 1 without using the conventional property at all. This property thus implies a definition of minimality on arbitrary SSA graphs, which implies cytron-minimality on conventional SSA form (as has already been noted by Braun et al.[1]).

## 4. Design and Implementation

Now that we've proven that the property of being free of redundant SCCs suffices for minimality, we turn our attention to the algorithm for establishing this property. In the following, we give an overview of the functional principles of this algorithm, give a bound for its algorithmic complexity, and detail the implementation of this algorithm created as part of this work.

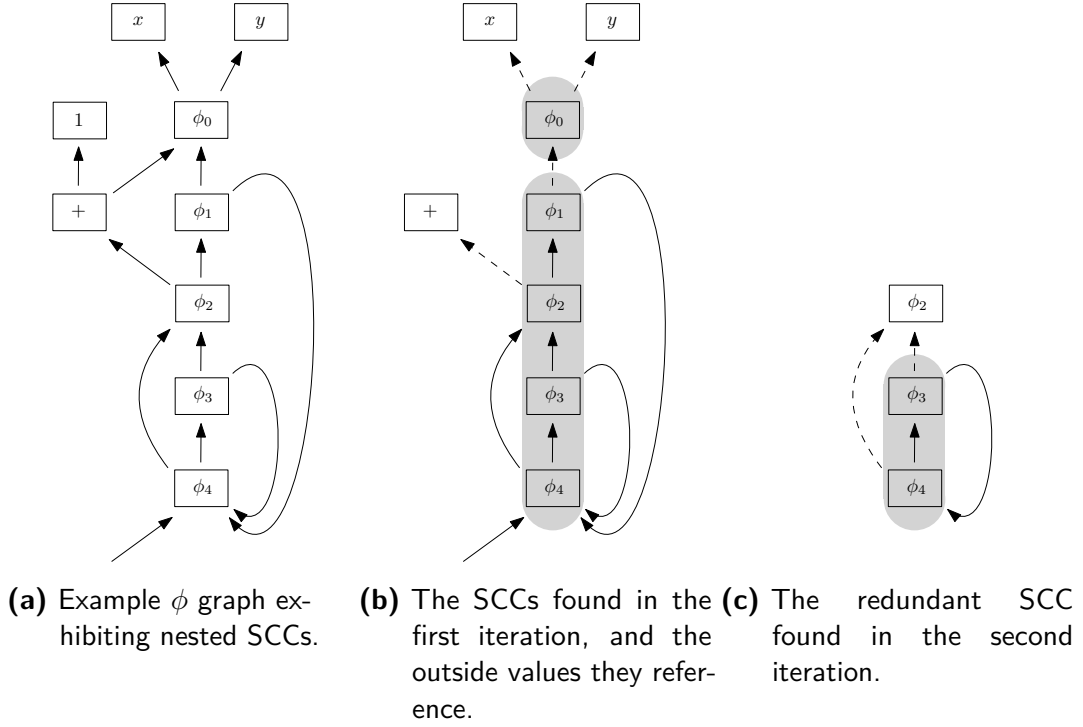
### 4.1. Braun et al.'s Peeling Algorithm

At a cursory glance, the problem of identifying all redundant SCCs might look like it could be solved using a single invocation of an SCC-finding algorithm. However, a redundant SCC may not be maximal, but might instead only be a maximal SCC on a subset of nodes. An example of a  $\phi$  graph exhibiting this behaviour can be seen in Figure 4.1.

#### 4.1.1. Description of the Algorithm

The algorithm proposed by Braun et al. for ensuring minimality works by recursively “peeling off” nodes which are guaranteed not to be in a redundant set. If a redundant SCC is found, all usage sites referring to nodes within this SCC are modified to instead use the true argument of the SCC. Since the removal of an SCC of redundant  $\phi$ -functions can render other sets of  $\phi$ -functions redundant, the algorithm processes the SCCs in reverse topological order to ensure that any redundant SCCs the current working set might have referred to have already been removed. “Processing an SCC” consists of checking whether all  $\phi$ -functions within it reference at most one common value outside the SCC (i.e. checking whether it is redundant). A representation of this algorithm in pseudocode can be seen in Figure 4.2.

Each recursive call can safely reduce the size of the working set, as those  $\phi$ -functions which have an argument outside the current SCC are guaranteed not to be included in any redundant SCC contained therein:



**Figure 4.1.:** An example from Braun et al. [1] illustrating the need for examining nested SCCs. Edges point from usage site to referenced value.

*Proof.* Suppose that such a node  $\varphi$  were part of a redundant SCC  $P' \subset P$ . Since  $P'$  is redundant, the sole value referred to by nodes within  $P'$  outside of itself must be the value  $\varphi$  is referring to. However, since  $P$  itself is strongly connected and  $P'$  is contained in  $P$ , there must be some node in  $P'$  which has an argument in  $P \setminus P'$ . This argument constitutes a second argument outside of  $P'$  different from the first, a contradiction that  $P'$  is redundant. Thus  $\varphi$  cannot be an element of a redundant SCC contained in  $P$ .  $\square$

This allows us to start with SCCs induced by all nodes (i.e. actual, maximal SCCs) and to gradually “peel off” nodes to reduce the problem to a smaller instance.

A natural choice for the algorithm used to compute the  $\phi$  SCCs is Tarjan’s Algorithm [10], as it not only computes the SCCs efficiently (in  $\mathcal{O}(|V| + |E|)$ ), but additionally produces them in reverse topological ordering, which elides the need for a separate sorting step.

```

1 def remove_redundant_phis(phis):
2     sccs = compute_phi_SCCs(induced_subgraph(phis))
3     for scc in topological_sorted(sccs):
4         process_scc(scc)
5
6 def process_scc(scc)
7     inner = set()
8     outer_ops = set()
9     for phi in scc:
10        is_inner = True
11        for operand in phi.get_operands():
12            if operand not in scc:
13                outer_ops.add(operand)
14                is_inner = False
15        if is_inner:
16            inner.add(phi)
17
18 if len(outer_ops) == 1:
19     # definite assignment gives us that there is at least one outer op.
20     replace_scc_by_value(scc, outer_ops.pop())
21 else
22     # len(outer_ops) must be >= 2, thus scc isn't redundant.
23     remove_redundant_phis(inner)

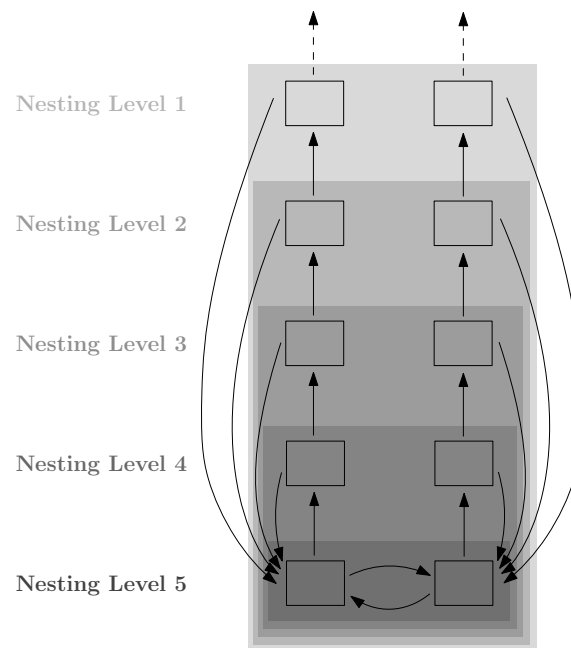
```

Figure 4.2.: Braun et al.'s algorithm for removing redundant phis

### 4.1.2. Algorithmic Complexity

The next step towards a complete understanding of this algorithm is an analysis of its runtime complexity. In this section we thus present a worst-case scenario for the algorithm and analyze its asymptotic behaviour to obtain a lower bound on the asymptotic complexity of the algorithm. We then motivate why no other graph can achieve a running time asymptotically worse than these graphs, completing the analysis.

First, it is worth noting that the act of replacing a redundant SCC by another value takes linear time in the order of usage sites of nodes within the SCC. However, since redundant SCCs are removed in reverse topological order, every edge in the graph is modified at most once, so we have an upper bound of  $\mathcal{O}(|E|)$  for this overhead. This overhead is unavoidable and in a certain sense work “outside” of the algorithm. As such, it does not figure in our complexity analysis. Since the algorithm handles  $\phi$  graphs and only comes into contact with the underlying SSA graph when removing SCCs, the complexity analysis reasons exclusively in terms of  $\phi$  graphs, and  $V$  and  $E$  denote the  $\phi$ -functions and  $\phi$  arguments, not the CFG nodes and edges.



**Figure 4.3.:** A schema for  $\phi$  graphs with arbitrarily deeply nested SCCs.

Figure 4.3 shows the type of graph we'll examine. These graphs can have an arbitrary number of nested SCCs, each only two nodes smaller than the previous. A graph with  $n$  such nesting steps has  $2n$  nodes, each of which has a well-defined nesting level (i.e. the nesting level of the innermost SCC the node still belongs to). A node with nesting level  $l$  in such a graph may only have edges to nodes in nesting levels greater or equal to  $l - 1$ . This leads to an upper bound on the number of edges in an  $n$ -nesting-level graph of  $4 \sum_{i=2}^{n+1} i = 2n^2 + 8n + 2$ . Stepping through an execution of the algorithm on such a graph, the first iteration of Tarjan's algorithm will find the maximal SCC, and the inner phase of the algorithm will then evaluate every edge within this SCC once. Since this SCC is not redundant, the two nodes referring to nodes outside of this SCC are removed from the working set. The remainder of the working set is now identical to an execution of the algorithm on an instance one step smaller. The work performed during such an execution with  $n$  steps of nested SCCs can thus be summarized by the following recurrence relation:

$$T(n) = T(n - 1) + \mathcal{O}(|V(n)| + |E(n)|) + |E(n)| = \sum_{i=1}^n (\mathcal{O}(|V(i)| + |E(i)|) + |E(i)|)$$

Evaluating this sum yields a runtime of  $\mathcal{O}(|V|^2 + |V||E|)$ . This gives us a lower bound on the complexity of this algorithm.

In fact, such a graph represents a true worst-case scenario for this algorithm: Consider the task of constructing a graph with  $n$  nodes to maximize execution time of this algorithm. Clearly, it is not beneficial to construct redundant sets, as no further time is spent on these. Furthermore, for a working set at a given recursion level to allow continued recursion, there must be at least two nodes that will be removed from the working set in the next iteration: Every step of recursion removes nodes pointing outside their SCC from the working set, so nodes within a working set only ever point to nodes that were still in the working set in the previous recursion level. This means that the two nodes referred to by SCC nodes in the next iteration to trigger more recursive steps will have to "come from" the working set of this iteration. Ultimately, this means a graph with  $n$  nodes can cause a maximum recursion depth of  $\frac{n}{2}$ .

Remember that Tarjan's algorithm only performs depth-first search within working sets, and the rest of the work per working set is bounded by the number of edges within that working set. Hence the amount of work done at a given level of recursion is only dependent on the combined size of the different working sets. Attempting to construct an execution with a high branching factor in its recursive calls is thus pointless, since every recursive call effectively reduces the working set size by at least two. Simply nesting SCCs in a linear fashion allows for the most recursive calls for a given number of nodes and thus for the most amount of work per node.

Thus the graph schema from Figure 4.3 truly is a worst-case scenario for the algorithm, and our complexity bound of  $\mathcal{O}(|V|^2 + |V||E|)$  is the asymptotic complexity of this algorithm.

## 4.2. Implementation

As part of this thesis, we’ve implemented Braun et al.’s peeling algorithm in libFIRM [11]. The algorithm is implemented as a pass to be run during SSA optimization. The implementation can be found in Appendix A.2 and on Github<sup>1</sup>. Since the efficiency of data structures is instrumental in ensuring that the implementation performs as well as the theoretical asymptotic bounds allow for, we list and motivate our choice of data structures.

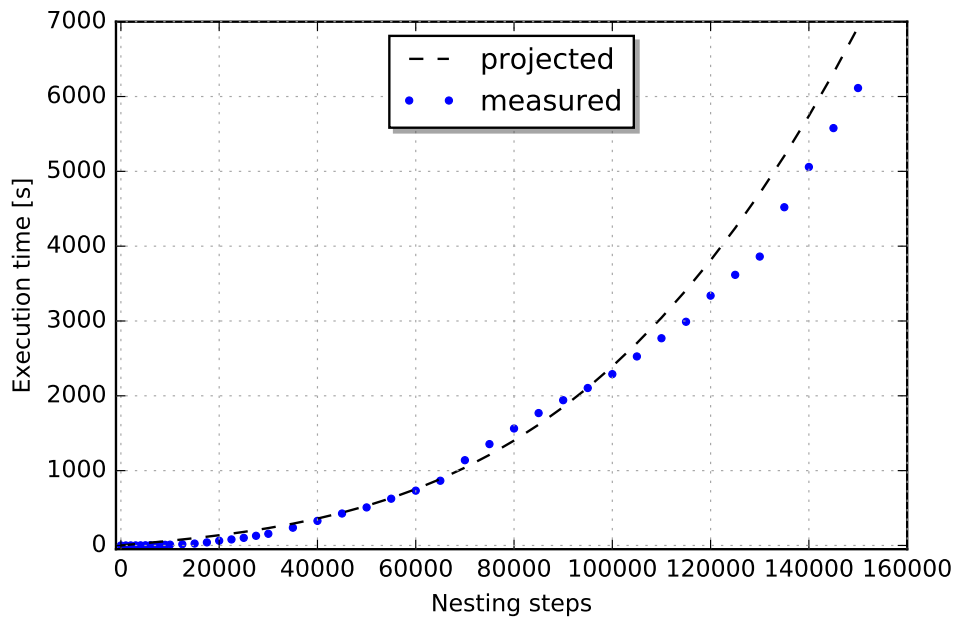
We use a hashmap to map redundant  $\phi$ -functions to the SSA values they should be replaced by, to defer modifications to the graph until the algorithm is done. SCCs store their set of nodes within an iterable hashset, since they provide efficient insertion, deletion, iteration and element tests. These operations are possible in *amortized*  $\mathcal{O}(1)$ , and are usually available in *average*  $\mathcal{O}(1)$ . The list of SCCs yet to be evaluated is stored in a doubly linked list, and we use another temporary doubly linked list for the SCCs found within one execution of Tarjan’s algorithm. At the end of one such execution, the new batch of SCCs are spliced to the front of the yet-to-be-evaluated-list, and every iteration only pops SCCs off from the front of this list. This mode of operation naturally meshes with the reverse topological ordering in which SCCs need to be processed. These operations can be done in  $\mathcal{O}(1)$ . Fortunately libFIRM provides implementations of all these data structures, so the entirety of the code that needed to be written encompasses around 400 lines of C.

One caveat not mentioned so far is that the  $\phi$  graph must first be constructed. Thus, if the CFG is orders of magnitude bigger than the  $\phi$  web, the execution time of the first iteration may be much greater than what one would expect given the number of  $\phi$ -functions. An implementation that is only used immediately after SSA construction could benefit from this by having the SSA construction pass also make note of all  $\phi$ -functions per variable, thus amortizing the construction costs for the  $\phi$  graph. Since we want to compare the efficacy of the peeling algorithm when run at different points in the optimization pipeline, our implementation doesn’t benefit from this. Instead, the  $\phi$  graph is never explicitly constructed; A first iteration of the algorithm is performed by walking the whole CFG, starting an execution of Tarjan’s algorithm from every  $\phi$ -function not yet handled. The remainder of the working set after this first iteration then includes only the  $\phi$ -functions which would otherwise be present at this stage.

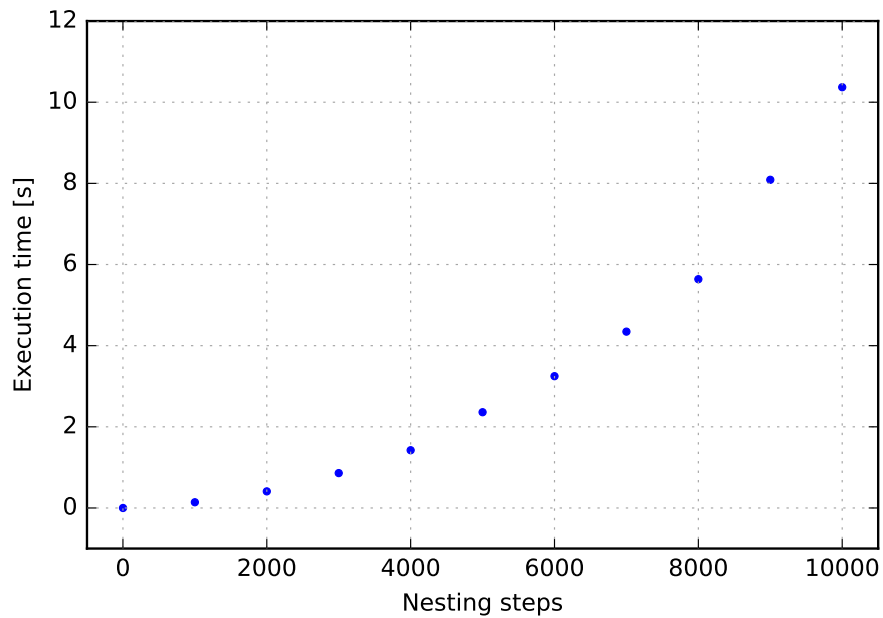
---

<sup>1</sup>[https://github.com/TehMillhouse/libfirm/tree/SSA\\_arbitrary\\_control\\_flow](https://github.com/TehMillhouse/libfirm/tree/SSA_arbitrary_control_flow)





**Figure 4.4.:** Execution time of our implementation on a constructed worst-case example. The dotted line shows the trajectory of our complexity bound fitted to the first two-thirds of the data range.



**Figure 4.5.:** Detailed crop of Figure 4.4.

To shed some light on the size at which  $\phi$  graphs become untractable for this implementation, we have tested the algorithm on a version of the worst-case graph from Figure 4.3. Note that since these graphs were constructed programmatically by directly calling FIRM node constructors, they don't correspond to the internal representation of any C program. However, this only really affects the first iteration of the algorithm, since the first iteration has to traverse all nodes in the IR to construct the initial working set. In any case, this "missing overhead" can be avoided in an implementation of the peeling algorithm by coupling the minimization step and the SSA construction, and having the SSA construction algorithm explicitly construct the  $\phi$  graph.<sup>2</sup> Figures 4.4 and 4.5 show the results of this informal test. In our experiments, which we more rigorously evaluate in Section 5, we have found no function with more than seven hundred  $\phi$ -functions in total. Since our implementation still completes in under a second on a worst-case instance with a similar number of  $\phi$ -functions *in a single  $\phi$  net*, we argue that the practical performance of both Braun et al.'s peeling algorithm, and specifically our implementation of it, are by all means adequate.

---

<sup>2</sup>We have not taken this approach as we want to evaluate the algorithm both when executed right after SSA construction and during SSA optimization

## 5. Evaluation

The phenomenon of irreducibility in CFGs is often associated with the `GOTO` statement. However, even languages lacking a direct equivalent to `GOTO` may have interactions of language constructs allowing for irreducible control flow to occur, Duff’s Device being the most famous example of such an interaction. Still, it has been noted that irreducible control flow is a rather rare occurrence, even in machine-generated code [12, 13]. It’s thus a valid question to ask whether these rare occurrences introduce enough redundant  $\phi$ -functions to warrant adding additional complexity in the form of Braun et al.’s peeling algorithm to an SSA-based compiler. To answer this question, we’ve evaluated the peeling algorithm as applied to a collection of real-world programs. Our input programs are taken from the SPEC CPU2000 and CPU2006 benchmark suites. Several real-world C projects are taken from the libFIRM test suite (i.e. the Apache web server v2.2.21, vim v7.2 and lua v5.1.4). In addition to this, we’ve included spass v2.1 and the reference implementation of the python language (cpython, v3.7 alpha @ 321fd5). We would have loved to include the Linux kernel into our evaluation, as its large codebase of low-level systems programming code makes it one of the “worst offenders” when it comes to the sheer number of `GOTOS`. Unfortunately, writing a compiler that can compile the Linux kernel is a herculean task, and currently only `gcc` is capable of doing so.

All measurements (including the measurements of figures 4.4 and 4.5) were performed on the same machine. This machine is running a standard Ubuntu 16.04.1 LTS (kernel version 4.4.0) and is equipped with 16GB of memory and an Intel i7-2600 processor clocked at 3.4GHz. Since trivial  $\phi$ -functions can also be eliminated using a much simpler, more efficient algorithm, we have specifically excluded them from our numbers by removing them before executing the peeling algorithm. All standard deviations given assume an underlying normal distribution.

### 5.1. Occurrence of Redundant $\phi$ -Functions

We start out by measuring the time spent during compilation (without linking) and during execution of the peeling algorithm, as well as the number of redundant  $\phi$ -functions found. This is done in two kinds of configurations: A baseline configuration,

with all non-essential optimizations turned off<sup>1</sup> and an optimized configuration with all conventional optimizations turned on. This is done to gauge multiple things:

- How often does irreducible control flow occur at source level?
- Do redundant  $\phi$ -functions occur in real-world situations as a result of SSA optimizations modifying the IR?
- How many of the redundant  $\phi$ -functions found at source level are “accidentally” optimized away as side-effect of other optimizations?
- How well does the peeling algorithm perform on real-world programs, and how does the time spent minimizing compare to the total compilation time?

Unfortunately, due to multiple preexisting bugs in the compiler we used, not all programs can be compiled with all optimizations turned off. In this case, we instead turn off a common set of optimizations to get as close as possible to our original goal<sup>2</sup>. Such datapoints are marked with an asterisk (\*) after the benchmark name.

Tables 5.1, 5.2 and 5.3 show our results in the fully optimized case. We concede that redundant  $\phi$ -functions seem to be rather rare, with SPEC CPU2000 exhibiting none and only two benchmarks from SPEC CPU2006 exhibiting any redundant  $\phi$ -functions. We note that minimization takes less than 0.05% of total compilation time in most cases, the only exception being the SPEC CPU2006 version of bzip2, where the peeling algorithm manages to remove almost a fifth of  $\phi$ -functions. The single benchmark with the highest number of redundant  $\phi$ -functions in this configuration is vim, with as many as 362 redundant  $\phi$ -functions in a single function.

Tables 5.4, 5.5 and 5.6 show the same benchmarks compiled with only essential optimizations and our minimization pass. In general, there tend to be more redundant  $\phi$ -functions in an unoptimized SSA graph, suggesting that at least some of the redundant  $\phi$ -functions present at source level can be found and removed by other means. We observe the greatest increase in redundant  $\phi$ -functions in cpython. This increase stems from a single function, `_PyEval_EvalFrameDefault`, which contains the core language interpreter loop. We have manually determined that libFIRM’s “combo” pass removes all redundant  $\phi$ -functions in this function. Since less time is spent on other optimizations, the peeling algorithm takes more time percentually than with all optimizations turned on. However, even in this configuration the time spent minimizing the SSA graph is well below 0.5% of total compilation time. We note the sharp decrease in redundant  $\phi$ -functions in vim when turning *off* optimizations.

---

<sup>1</sup>libFIRM requires a certain number of basic optimizations to be turned on to generate machine code.

<sup>2</sup>This set consists of loop strength reduction, jump threading, global common subexpression elimination, and the combined dataflow analysis phase based on the work of Click [14].

benchmark	time peeling	st. dev.	total time	st. dev.	% of total time	$\phi$ s removed
vortex	47.09 ms	0.30 %	115.30 s	0.26 %	0.041 %	0/1583
bzip2	2.86 ms	1.33 %	9.42 s	0.72 %	0.030 %	0/324
vpr	11.95 ms	0.81 %	33.57 s	0.45 %	0.036 %	0/978
crafty	17.30 ms	1.62 %	60.66 s	1.25 %	0.029 %	0/1111
gcc	153.70 ms	0.21 %	484.40 s	0.25 %	0.032 %	0/9119
gzip	4.42 ms	0.70 %	12.19 s	0.75 %	0.036 %	0/472
perlbnk	63.40 ms	0.30 %	201.30 s	0.29 %	0.031 %	0/3858
mesa	50.24 ms	0.24 %	135.10 s	0.30 %	0.037 %	0/3409
ampp	13.71 ms	0.44 %	29.24 s	0.58 %	0.047 %	0/1058
gap	62.11 ms	0.27 %	164.80 s	0.16 %	0.038 %	0/6570
mcf	1.54 ms	1.75 %	3.31 s	1.51 %	0.047 %	0/112
twolf	26.23 ms	0.26 %	74.05 s	0.30 %	0.035 %	0/2154
parser	10.92 ms	0.49 %	42.05 s	0.33 %	0.026 %	0/1136
equake	1.83 ms	0.37 %	3.94 s	0.74 %	0.046 %	0/161
art	1.29 ms	0.77 %	3.11 s	1.32 %	0.042 %	0/156

**Table 5.1.:** Performance and efficacy of the peeling algorithm alongside a full set of optimizations in SPEC CPU2000. Standard deviations assume a normal distribution and are listed alongside their corresponding data point. The measured mean times correspond to the time spent executing the peeling algorithm and the total compilation time, respectively. The rightmost column lists the number of redundant  $\phi$ -functions that were eliminated and the total number of  $\phi$ -functions in the benchmark before minimization.

benchmark	time peeling	st. dev.	total time	st. dev.	% of total time	$\phi$ s removed
libquantum	3.18 ms	0.62 %	11.72 s	0.54 %	0.027 %	0/211
perlbench	145.20 ms	0.51 %	442.60 s	2.16 %	0.033 %	30/7626
bzip2	52.33 ms	1.37 %	21.09 s	0.41 %	0.250 %	187/965
milc	11.35 ms	1.15 %	33.37 s	0.54 %	0.034 %	0/966
mcf	1.56 ms	2.08 %	3.41 s	2.60 %	0.046 %	0/111
h264ref	56.86 ms	0.17 %	178.59 s	1.83 %	0.032 %	0/4667
sphinx3	16.00 ms	0.82 %	40.50 s	0.57 %	0.039 %	0/1364
lbm	1.02 ms	0.74 %	2.09 s	0.35 %	0.049 %	0/60
sjeng	11.73 ms	0.36 %	34.66 s	0.35 %	0.034 %	0/750
gcc	340.80 ms	0.20 %	1375.04 s	0.77 %	0.025 %	0/16624
hmmer	28.84 ms	0.26 %	79.38 s	0.08 %	0.036 %	0/2187
gobmk	73.89 ms	0.09 %	223.61 s	0.18 %	0.033 %	0/4620

**Table 5.2.:** Performance and efficacy of the peeling algorithm alongside a full set of optimizations in SPEC CPU2006. Standard deviations assume a normal distribution and are listed alongside their corresponding data point. The measured mean times correspond to the time spent executing the peeling algorithm and the total compilation time, respectively. The rightmost column lists the number of redundant  $\phi$ -functions that were eliminated and the total number of  $\phi$ -functions in the benchmark before minimization.

benchmark	time peeling	st. dev.	total time	st. dev.	% of total time	$\phi$ s removed
spass	68.26 ms	0.48 %	260.9 s	0.41 %	0.026 %	0/5178
apache	38.46 ms	96.49 %	77.7 s	91.54 %	0.049 %	44/5382
lua	7.11 ms	53.03 %	21.4 s	95.34 %	0.033 %	0/802
vim	38.77 ms	0.39 %	103.7 s	0.21 %	0.037 %	449/13171
cpython	77.77 ms	1.05 %	203.5 s	0.43 %	0.038 %	90/17352

**Table 5.3.:** Performance and efficacy of the peeling algorithm alongside a full set of optimizations in misc. benchmarks. Standard deviations assume a normal distribution and are listed alongside their corresponding data point. The measured mean times correspond to the time spent executing the peeling algorithm and the total compilation time, respectively. The rightmost column lists the number of redundant  $\phi$ -functions that were eliminated and the total number of  $\phi$ -functions in the benchmark before minimization.

benchmark	time peeling	st. dev.	total time	st. dev.	% of time	$\phi$ s removed
vortex	27.17 ms	2.33 %	11.36 s	1.65 %	0.24 %	0/2727
bzip2	1.50 ms	10.04 %	0.47 s	6.03 %	0.31 %	0/416
vpr	7.94 ms	2.40 %	2.46 s	1.89 %	0.32 %	0/2923
crafty	11.31 ms	2.78 %	7.88 s	1.15 %	0.14 %	0/1437
gcc	94.50 ms	1.83 %	58.25 s	0.44 %	0.16 %	0/18491
gzip	2.83 ms	5.25 %	0.93 s	4.50 %	0.30 %	0/602
perlbmk	32.75 ms	2.61 %	16.29 s	0.44 %	0.20 %	0/6839
mesa	30.14 ms	2.41 %	11.16 s	1.89 %	0.27 %	0/5092
ammp	7.66 ms	3.05 %	2.71 s	2.89 %	0.28 %	0/1000
gap	39.53 ms	2.35 %	11.55 s	1.03 %	0.34 %	0/8493
mcf	0.67 ms	15.34 %	0.39 s	13.86 %	0.17 %	0/145
twolf	17.27 ms	3.95 %	7.83 s	3.10 %	0.22 %	0/2837
parser	6.05 ms	2.38 %	2.20 s	1.95 %	0.28 %	0/1312
equake	1.04 ms	9.93 %	0.34 s	7.31 %	0.31 %	0/179
art	0.64 ms	14.27 %	0.20 s	11.66 %	0.33 %	0/217

**Table 5.4.:** Performance and efficacy of the peeling algorithm in SPEC CPU2000 benchmarks when using only essential optimizations. Standard deviations assume a normal distribution and are listed alongside their corresponding data point. The measured mean times correspond to the time spent executing the peeling algorithm and the total compilation time, respectively. The rightmost column lists the number of redundant  $\phi$ -functions that were eliminated and the total number of  $\phi$ -functions in the benchmark before minimization.

Interestingly, there are no functions in the vim code base which exhibit redundant  $\phi$ -functions in both configurations. The vim benchmark thus shows two kinds of interesting behaviour simultaneously: redundant  $\phi$ -functions being removed by other optimizations and sets of  $\phi$ -functions being made redundant by other optimizations. Through manual analysis, we have determined that the combo phase is the cause of almost all of the redundant  $\phi$ -functions observed in vim in the optimized configuration.

## 5.2. Effect of Redundant $\phi$ -Functions on Performance

In this section we evaluate the effects of redundant  $\phi$ -functions on the efficiency of the generated code. That is, we attempt to answer the question “But does it even matter?”. Unfortunately, the persuasiveness of our results is limited, since we

benchmark	time peeling	st. dev.	total time	st. dev.	% of time	$\phi$ s removed
libquantum	1.62 ms	4.83 %	0.92 s	4.87 %	0.18 %	0/268
perlbench	62.18 ms	1.51 %	69.90 s	0.26 %	0.09 %	226/11821
bzip2	4.99 ms	5.80 %	1.84 s	3.05 %	0.27 %	401/1243
milc	5.71 ms	6.71 %	2.53 s	3.6 %	0.23 %	0/983
mcf	0.73 ms	5.40 %	0.42 s	4.58 %	0.17 %	0/148
h264ref	35.89 ms	2.25 %	28.13 s	0.51 %	0.13 %	7/5884
sphinx3	8.12 ms	1.98 %	3.91 s	2.25 %	0.21 %	0/1376
lbm	1.29 ms	2.62 %	0.52 s	3.23 %	0.25 %	0/60
sjeng	7.80 ms	5.99 %	4.41 s	2.46 %	0.18 %	0/1190
gcc	180.90 ms	2.86 %	247.70 s	0.16 %	0.073 %	0/35571
hmmer	16.45 ms	1.86 %	7.12 s	1.77 %	0.23 %	0/2491
gobmk	36.81 ms	1.77 %	24.32 s	0.67 %	0.15 %	0/6217

**Table 5.5.:** Performance and efficacy of the peeling algorithm in SPEC CPU2006 benchmarks when using only essential optimizations. Standard deviations assume a normal distribution and are listed alongside their corresponding data point. The measured mean times correspond to the time spent executing the peeling algorithm and the total compilation time, respectively. The rightmost column lists the number of redundant  $\phi$ -functions that were eliminated and the total number of  $\phi$ -functions in the benchmark before minimization.

benchmark	time peeling	st. dev.	total time	st. dev.	% of total time	$\phi$ s removed
spass	31.13 ms	2.39 %	15.46 s	1.50 %	0.20 %	0/6388
apache	31.10 ms	1.86 %	27.06 s	1.23 %	0.11 %	63/7051
lua*	6.11 ms	0.87 %	11.22 s	0.51 %	0.05 %	0/813
vim*	70.77 ms	1.46 %	30.80 s	0.58 %	0.23 %	31/16874
cpython*	87.23 ms	0.73 %	113.01 s	0.57 %	0.08 %	1055/19973

**Table 5.6.:** Performance and efficacy of the peeling algorithm in misc. benchmarks when using only essential optimizations. Standard deviations assume a normal distribution and are listed alongside their corresponding data point. The measured mean times correspond to the time spent executing the peeling algorithm and the total compilation time, respectively. The rightmost column lists the number of redundant  $\phi$ -functions that were eliminated and the total number of  $\phi$ -functions in the benchmark before minimization.



benchmark	minimized	st. dev.	not minimized	st. dev.	improvement
bzip2	490.39 s	0.38 %	492.64 s	0.36 %	0.46 %
h264ref	771.72 s	0.66 %	767.64 s	0.15 %	-0.53 %
perlbench	810.68 s	0.11 %	811.19 s	0.15 %	0.06 %
cpython (fannkuch.py)	201.45 s	0.49 %	205.31 s	0.96 %	1.88 %
cpython (nqueens.py)	32.11 s	0.83 %	32.51 s	0.53 %	1.25 %

**Table 5.7.:** Effect of redundant  $\phi$  elimination on benchmark run time when using only essential optimizations. “minimized” and “not minimized” times refer to whether the peeling algorithm has been enabled (minimized) or not (not minimized). Standard deviations assume a normal distribution and are listed alongside their corresponding data point.

have found very few programs exhibiting redundant  $\phi$ -functions. As an additional hurdle, due to time constraints, we could not measure performance data for programs outside the SPEC benchmark suites and cpython. For performance benchmarking of cpython, we adopted two benchmarks used by the PyPy project<sup>3</sup>. These two were simply chosen because they are relatively well-known and used in various forms across a large number of languages.

Tables 5.7 and 5.8 list the performance measurements we performed. To our surprise, the difference in performance was measurable and statistically significant in the case of the version of bzip2 included in SPEC CPU2006 and in cpython. Puzzlingly, the baseline configuration of bzip2, which showed a higher number of redundant  $\phi$ -functions, did not benefit as much from redundant  $\phi$ -function removal as the optimized configuration. We suspect this stems from the fact that a more concise SSA representation may enable other optimizations. In this sense, redundant  $\phi$ -functions only serve to “mask” patterns that other optimizations may match for. This result proves that, while rare, there are indeed cases where redundant  $\phi$ -functions can have a measurable performance impact on the generated code.

### 5.3. Sources of Redundant $\phi$ -Functions

We have manually inspected each function with at least ten redundant  $\phi$ -functions across all benchmarks to attempt to spot some of the more common patterns in code which exhibits redundant  $\phi$ -functions. Table 5.9 shows a summary of our results. Note that there are several functions with different purposes that only express redundant  $\phi$ -functions if other optimizations are included in the compilation

<sup>3</sup>See <https://bitbucket.org/py/py/benchmarks> for the original sources.

benchmark	minimized	st. dev.	not minimized	st. dev.	improvement
perlbench	366.43 s	0.48 %	366.69 s	0.22 %	0.07 %
bzip2	397.20 s	0.36 %	404.54 s	0.55 %	1.80 %

**Table 5.8.:** Effect of redundant  $\phi$  elimination on benchmark run time when using all optimizations. cpython, apache and vim are omitted for lack of benchmarks that execute code with redundant  $\phi$ -functions. “minimized” and “not minimized” times refer to whether the peeling algorithm has been enabled (minimized) or not (not minimized). Standard deviations assume a normal distribution and are listed alongside their corresponding data point.

benchmark	function name	min. opt.	full opt.	purpose
perlbench	Perl_re_intuit_start	✓	✓	regex engine
	Perl_ck_subr	✓	✓	interpreter
	Perl_regexec_flags	✓		regex engine
	Perl_sv_vcatpvfn	✓		string formatter
bzip2	unRLE_obuf_to_output_FAST	✓		?
	BZ2_decompress	✓	✓	decompressor
apache	apr_socket_sendfile	✓		network code
	apr_socket_sendv	✓		network code
	apr_vformatter		✓	string formatter
	fnmatch_ch	✓		pattern matching
	compile_branch		✓	regex engine
vim	vgetorpeek		✓	input routine
	win_line		✓	display text
	find_pattern_in_path	✓		pattern matching
cpython	sre_ucsX_match	✓	✓	regex engine
	_PyEval_EvalFrameDefault	✓		interpreter

**Table 5.9.:** All functions that, in some configuration, expressed ten or more redundant  $\phi$ -functions. The columns labeled “min. opt.” and “full opt.” list whether redundant  $\phi$ -functions were found in the baseline and fully optimized configurations, respectively.

process. However, it is hard to miss that among these functions, certain kinds of software occur suspiciously often: regular expression engines, string formatting (and other kinds of ad-hoc parsers), and the inner loops of language interpreters tend to generate redundant  $\phi$ -functions. In fact, these types of functions, along with the bzip2 decompressor and vim's input handling, have one thing in common. They all perform a form of dynamic dispatch, that is, they all contain a loop iterating over some input sequence and decide based on the current item which operation to perform.

Naive implementations of such a dispatch have a well-documented disadvantage: The branch misprediction penalty incurred at the dispatch location due to the inherent unpredictability of the operation that is to be performed. A common mitigation for this overhead is to avoid having a single dispatch point by using GOTOS to manually thread the flow of execution. Most of the functions in Table 5.9 employ this kind of manual threading.

There are, however, exceptions to this. The network code in apache has GOTOS leading into a loop, and is uninteresting for our purposes. The `win_line` function is used by vim to render a single line of text into the window buffer. It is especially noteworthy because even after macro expansion, it contains no GOTOS and no interlacing of control flow structures à la Duff's Device. In spite of this, it generates about as many redundant  $\phi$ -functions (362) as non-redundant ones (410) when compiled alongside libFIRM's combo pass. This provides an excellent example for the fact that even without irreducible control flow at source level, redundant  $\phi$ -functions can occur.



## 6. Related Work

Interactive theorem provers have grown powerful and useful enough that even daunting tasks such as constructing fully verified compilers have become tractable. It is thus that recent years have seen an uptick in academic publications in this field. This section serves to highlight some of these publications and put this thesis in its wider context.

Starting with SSA construction in general, Braun et al. [1] proposed an algorithm combining relative simplicity (forgoing the need for previous analyses) with performance which is competitive to that of LLVM’s implementation of the algorithm by Sreedhar et al. [15]. Buchwald et al. [4] then proceeded to implement a functional variant of Braun et al.’s SSA construction algorithm in Isabelle and proved the correctness of the algorithm and the minimality of its output given reducible CFGs, forming the framework this thesis is based upon.

Vellvm [16] is an implementation of LLVM’s IR in the theorem prover Coq. Recently, Zhao et al. [17] have formalized a subset of the IR supported by Vellvm called Vminus along with an operational semantic for it. They then proceeded to implement LLVM’s `mem2reg` SSA transformation pass in Vminus and prove it correct. Our approach differs in that our framework does not formalize a direct semantic for SSA, but concerns itself with semantic equivalence between an input CFG and its SSA version. Following this, Zhao et al. [18] then formalized the notion of dominance in this framework. They used Vminus to implement several algorithms for dominance analysis and proved each of them correct. They stated construction of dominator trees (which enable several SSA optimizations) and an SSA type checker as possible applications of their work. While still non-obvious, they thereby proved that an efficient implementation of dominance analysis is possible within the scope of verified compiler construction.

CompCert [19] is an unrelated project aiming to write a completely verified compiler for a substantial subset of the C language. CompCert is also implemented and verified in Coq, but unlike Vellvm, it uses a register transfer language as IR. Recent developments include an SSA-based middle-end [20] given the name CompCertSSA. Buchwald et al. [4] extracted their implementation of Braun et al.’s SSA construction algorithm into a form usable by CompCertSSA, providing a second SSA middle-end which is directly verified, as opposed to the translation validation approach taken by

---

CompCertSSA.

In the meantime, Demange et al. [21] have turned their attention to implementing and verifying SSA-based optimizations (specifically, Global Value Numbering and Sparse Conditional Constant Propagation) to make better use of their SSA middle-end for CompCert. Another interesting development is the work by Demange et al. [22] on extending CompCertSSA by improving the previous SSA destruction phase. Their contribution is an implementation and verification of the SSA destruction algorithm of Boissinot et al. [23], thus enabling code generation for a larger class of SSA CFGs and improving compiler output significantly.

In more generally related papers, Mansky et al. [24] provided a small-step operational semantic for CFGs and a variant of TRANS, a graph transformation language, to prove the correctness of an SSA construction algorithm in Isabelle/HOL. Blech et al. [25] provided a formalization of the semantics of SSA form and of a simple machine language. They proved that every translation from SSA to machine language that respects the partial ordering induced by use-def-edges in the SSA IR is a valid scheduling. They thus provided an example for how even the later stages of compilation can be made formally tractable.

## 7. Conclusion

Using the theorem prover Isabelle/HOL, we formally proved the correctness of Braun et al.’s peeling algorithm. We then proved that freedom of redundant  $\phi$ -functions implies a criterion for minimality that, in the case of *conventional* SSA, coincides with cytron-minimality. This definition of minimality is thus more general than cytron-minimality, and can be used to assess SSA form that has been modified by other optimizations. The entirety of the Isabelle proofs consists of around 1000 lines of proof text. We established a runtime complexity of  $\mathcal{O}(|V|^2 + |V||E|)$ , where  $V$  and  $E$  are the vertices and edges of the graph induced by  $\phi$ -functions and their arguments, respectively.

We implemented the algorithm as an optimization pass in libFIRM. The implementation consists of around 400 lines of C code, not counting reused data structure implementations that are part of libFIRM. Our experiments on worst-case instances of input CFGs suggest this implementation performs well enough on  $\phi$  webs of reasonable size.

In our evaluation of the algorithm, we vetted Braun et al.’s peeling algorithm against real-world programs from several benchmark suites and miscellaneous sources. These experiments confirm our previous judgment of the implementation’s performance on realistic programs. Our results further show that irreducible control flow is a rather rare occurrence, and program constructs that introduce redundant  $\phi$ -functions even more so. Nevertheless, such cases do exist. We found cases in which redundant  $\phi$ -functions could be optimized away by other analyses, as well as cases in which no other optimization in libFIRM could identify and remove these  $\phi$ -functions. We further found cases in which executing Braun et al.’s peeling algorithm along with other optimization passes uncovered redundant  $\phi$ -functions which were not present when executed with only minimal optimization after SSA construction. This suggests that there are situations in which even a compiler that guarantees construction of minimal SSA form could benefit from employing Braun et al.’s peeling algorithm as an optimization pass.

Using manual analysis of the functions exhibiting redundant  $\phi$ -functions at source level, we identified a common factor in these functions: All functions we found to be generating a substantial number of such  $\phi$ -functions were manually optimized interpreter loops doing some form of dynamic dispatch. These can be found in a wide

---

array of software, e.g. in implementations of programming languages, compression software, regular expression engines, emulators, and ad-hoc parsers. The redundant  $\phi$ -functions observed in these kinds of loops are a direct result of a common technique for mitigating branch misprediction overhead: manually threading control flow using GOTOS. The effect of large a number of redundant  $\phi$ -functions on the runtime of the generated code turned out to be statistically significant, but not large percentually.

We thus conclude that though there are cases where redundant  $\phi$ -functions can have a measurable impact on the performance of the generated code, in general these occur only rarely. Compilers which already use sophisticated data flow analyses such as that proposed by Click [14] can easily “get away” with ignoring redundant  $\phi$ -functions. In such cases Braun et al.’s SSA construction algorithm has proven itself to be a good compromise between implementation complexity, efficiency and quality of the output SSA form. The fact that redundant  $\phi$ -functions tend to occur in especially “hot” sections of code that have been hand-tuned for performance suggests that the added complexity may still be worth it in rare cases.

Though we proved the correctness and efficacy of Braun et al.’s peeling algorithm, the implementation we provide remains unverified, and we have encountered multiple bugs during implementation. Thus implementing a functional variant of the algorithm in Isabelle and proving it correct would be an obvious opportunity for future work. This could be combined into the effort by Buchwald et al. [4] to produce a fully verified SSA middle-end for CompCertSSA.



# Bibliography

- [1] M. Braun, S. Buchwald, S. Hack, R. Leißa, C. Mallon, and A. Zwinkau, “Simple and efficient construction of static single assignment form,” in *Compiler Construction* (R. Jhala and K. Bosschere, eds.), vol. 7791 of *Lecture Notes in Computer Science*, pp. 102–122, Springer, 2013.
- [2] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 451–490, Oct. 1991.
- [3] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, vol. 2283 of *LNCS*. Springer, 2002.
- [4] S. Buchwald, D. Lohner, and S. Ullrich, “Verified construction of static single assignment form,” in *25th International Conference on Compiler Construction* (M. Hermenegildo, ed.), CC 2016, pp. 67–76, ACM, 2016.
- [5] V. C. Sreedhar, R. D.-C. Ju, D. M. Gillies, and V. Santhanam, “Translating out of static single assignment form,” in *International Static Analysis Symposium*, pp. 194–210, Springer, 1999.
- [6] S. Ullrich and D. Lohner, “Verified construction of static single assignment form,” *Archive of Formal Proofs*, Feb. 2016. Formal proof development.
- [7] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc., 1986.
- [8] M. Wagner and D. Lohner, “Minimal construction of static single assignment form,” *Archive of Formal Proofs*, Nov. 2016. Formal proof development, to appear.
- [9] B. Nordhoff and P. Lammich, “Dijkstra’s shortest path algorithm,” *Archive of Formal Proofs*, Jan. 2012. Formal proof development.

- [10] R. Tarjan, “Depth first search and linear graph algorithms,” *SIAM Journal on Computing*, 1972.
- [11] G. Lindenmaier, “libFIRM – a library for compiler optimization research implementing FIRM,” Tech. Rep. 2002-5, Sept. 2002.
- [12] J. Stanier and D. Watson, “A study of irreducibility in c programs,” *Software: Practice and Experience*, vol. 42, no. 1, pp. 117–130, 2012.
- [13] D. E. Knuth, “An empirical study of fortran programs,” *Software: Practice and Experience*, vol. 1, no. 2, pp. 105–133, 1971.
- [14] C. N. Click, Jr., *Combining Analyses, Combining Optimizations*. PhD thesis, Houston, TX, USA, 1995.
- [15] V. C. Sreedhar and G. R. Gao, “A linear time algorithm for placing  $\phi$ -nodes,” in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL ’95, pp. 62–73, ACM, 1995.
- [16] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic, “Formalizing the LLVM intermediate representation for verified program transformations,” in *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL ’12, pp. 427–440, ACM, 2012.
- [17] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic, “Formal verification of SSA-based optimizations for LLVM,” in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI ’13, pp. 175–186, ACM, 2013.
- [18] J. Zhao and S. Zdancewic, “Mechanized verification of computing dominators for formalizing compilers,” in *Proceedings of the Second International Conference on Certified Programs and Proofs*, CPP’12, (Berlin, Heidelberg), pp. 27–42, Springer-Verlag, 2012.
- [19] X. Leroy, “A formally verified compiler back-end,” *J. Autom. Reason.*, vol. 43, pp. 363–446, Dec. 2009.
- [20] G. Barthe, D. Demange, and D. Pichardie, “A formally verified ssa-based middle-end: Static single assignment meets compcert,” in *Proceedings of the 21st European Conference on Programming Languages and Systems*, ESOP’12, (Berlin, Heidelberg), pp. 47–66, Springer-Verlag, 2012.
- [21] D. Demange, D. Pichardie, and L. Stefanescu, “Verifying Fast and Sparse SSA-Based Optimizations in Coq,” in *Proceedings of the 24th International*

- Conference on Compiler Construction*, vol. 9031 of *Lecture Notes in Computer Science*, pp. 233–252, Springer International Publishing, 2015.
- [22] D. Demange and Y. Fernandez de Retana, “Mechanizing conventional SSA for a verified destruction with coalescing,” in *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, (New York, NY, USA), pp. 77–87, ACM, 2016.
- [23] B. Boissinot, A. Darté, F. Rastello, B. D. de Dinechin, and C. Guillon, “Revisiting out-of-SSA translation for correctness, code quality and efficiency,” in *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO ’09, (Washington, DC, USA), pp. 114–125, IEEE Computer Society, 2009.
- [24] W. Mansky and E. Gunter, “A framework for formal verification of compiler optimizations,” in *Interactive Theorem Proving*, vol. 6172 of *Lecture Notes in Computer Science*, pp. 371–386, Springer, 2010.
- [25] J. O. Blech and S. Glesner, “A formal correctness proof for code generation from SSA form in Isabelle/HOL,” in *Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e. V. (GI)*, 2004.



# A. Appendices

## A.1. Full Isabelle Proof Document

### Minimality under Irreducible Control Flow

Braun et al. [1] provide an extension to the original construction algorithm to ensure minimality according to Cytron's definition even in the case of irreducible control flow. This extension establishes the property of being *redundant-scc-free*, i.e. the resulting graph  $G$  contains no subsets inducing a strongly connected subgraph  $G'$  via  $\phi$ -functions such that  $G'$  has less than two  $\phi$  arguments in  $G \setminus G'$ . In this section we will show that a graph with this property is Cytron-minimal.

```
theory Irreducible
imports Minimality Graph-path
begin
```

```
context CFG-SSA-Transformed
begin
```

#### Proof of Lemma 1 from Braun et al.

To preserve readability, we won't distinguish between graph nodes and the  $\phi$ -functions contained inside such a node.

The graph induced by the  $\phi$  network contained in the vertex set  $P$ . Note that the edges of this graph are not necessarily a subset of the edges of the input graph.

**definition** *induced-phi-graph*  $g P \equiv \{(\varphi, \varphi'). \text{ phiArg } g \ \varphi \ \varphi'\} \cap P \times P$

For the purposes of this section, we define a "redundant set" as a nonempty set of  $\phi$ -functions with at most one  $\phi$  argument outside itself. A redundant SCC is defined analogously. Note that since any uses of values in a redundant set can be replaced by uses of its singular argument (without modifying program semantics), the name is adequate.

**definition** *redundant-set*  $g P \equiv P \neq \{\} \wedge P \subseteq \text{dom } (\text{phi } g) \wedge (\exists v' \in \text{allVars } g. \forall \varphi \in P. \forall \varphi'. \text{ phiArg } g \ \varphi \ \varphi' \longrightarrow \varphi' \in P \cup \{v'\})$

**definition** *redundant-scc*  $g P$   $scc \equiv \text{redundant-set } g \text{ scc} \wedge \text{is-scc } (\text{induced-phi-graph } g P)$   
*scc*

We prove an important lemma via condensation graphs of  $\phi$  networks, so the relevant definitions are introduced here.

**definition** *condensation-nodes*  $g P \equiv \text{scc-of } (\text{induced-phi-graph } g P) \text{ ' } P$

**definition** *condensation-edges*  $g P \equiv ((\lambda(x,y). (\text{scc-of } (\text{induced-phi-graph } g P) x, \text{scc-of } (\text{induced-phi-graph } g P) y)) \text{ ' } (\text{induced-phi-graph } g P)) - \text{Id}$

For a finite  $P$ , the condensation graph induced by  $P$  is finite and acyclic.

**lemma** *condensation-finite*:  $\text{finite } (\text{condensation-edges } g P)$

The set of edges of the condensation graph, spanning at most all  $\phi$  nodes and their arguments (both of which are finite sets), is finite itself.

**proof** –

let  $?phiEdges = \{(a,b). \text{phiArg } g a b\}$

have  $\text{finite } ?phiEdges$

**proof** –

let  $?phiDomRan = (\text{dom } (\text{phi } g) \times \bigcup (\text{set ' } (\text{ran } (\text{phi } g))))$

from  $\text{phi-finite}$

have  $\text{finite } ?phiDomRan$  **by** ( $\text{simp add: imageE phi-finite map-dom-ran-finite}$ )

have  $?phiEdges \subseteq ?phiDomRan$

**apply** ( $\text{rule subst[of } \forall a \in ?phiEdges. a \in ?phiDomRan]$ )

**apply** ( $\text{simp-all add: subset-eq[symmetric] phiArg-def}$ )

**by** ( $\text{auto simp: ran-def}$ )

**with** ( $\text{finite } ?phiDomRan$ )

**show**  $\text{finite } ?phiEdges$  **by** ( $\text{rule Finite-Set.rev-finite-subset}$ )

**qed**

**hence**  $\bigwedge f. \text{finite } (f \text{ ' } (?phiEdges \cap (P \times P)))$  **by**  $\text{auto}$

**thus**  $\text{finite } (\text{condensation-edges } g P)$  **unfolding**  $\text{condensation-edges-def induced-phi-graph-def}$   
**by**  $\text{auto}$

**qed**

auxiliary lemmas for acyclicity

**lemma** *condensation-nodes-edges*:  $(\text{condensation-edges } g P) \subseteq (\text{condensation-nodes } g P \times \text{condensation-nodes } g P)$

**unfolding**  $\text{condensation-edges-def condensation-nodes-def induced-phi-graph-def}$

**by**  $\text{auto}$

**lemma** *condensation-edge-impl-path*:

**assumes**  $(a, b) \in (\text{condensation-edges } g P)$

**assumes**  $(\varphi_a \in a)$

**assumes**  $(\varphi_b \in b)$   
**shows**  $(\varphi_a, \varphi_b) \in (\text{induced-phi-graph } g \ P)^*$   
**unfolding** *condensation-edges-def*  
**proof** –  
**from** *assms(1)*  
**obtain**  $x \ y$  **where** *x-y-props*:  
 $(x, y) \in (\text{induced-phi-graph } g \ P)$   
 $a = \text{scc-of } (\text{induced-phi-graph } g \ P) \ x$   
 $b = \text{scc-of } (\text{induced-phi-graph } g \ P) \ y$   
**unfolding** *condensation-edges-def* **by** *auto*  
**hence**  $x \in a \ y \in b$  **by** *auto*

All that's left is to combine these paths.

**with** *assms(2) x-y-props(2)*  
**have**  $(\varphi_a, x) \in (\text{induced-phi-graph } g \ P)^*$  **by** *(meson is-scc-connected scc-of-is-scc)*  
**moreover with** *assms(3) x-y-props(3) (y \in b)*  
**have**  $(y, \varphi_b) \in (\text{induced-phi-graph } g \ P)^*$  **by** *(meson is-scc-connected scc-of-is-scc)*  
**ultimately**  
**show**  $(\varphi_a, \varphi_b) \in (\text{induced-phi-graph } g \ P)^*$  **using** *x-y-props(1)* **by** *auto*  
**qed**

**lemma** *path-in-condensation-impl-path*:

**assumes**  $(a, b) \in (\text{condensation-edges } g \ P)^+$   
**assumes**  $(\varphi_a \in a)$   
**assumes**  $(\varphi_b \in b)$   
**shows**  $(\varphi_a, \varphi_b) \in (\text{induced-phi-graph } g \ P)^*$   
**using** *assms*  
**proof** *(induction arbitrary: \varphi\_b rule:trancl-induct)*  
**fix**  $y \ z \ \varphi_b$   
**assume**  $(y, z) \in \text{condensation-edges } g \ P$

**hence** *is-scc*  $(\text{induced-phi-graph } g \ P) \ y$  **unfolding** *condensation-edges-def* **by** *auto*  
**hence**  $\exists \varphi_y. \varphi_y \in y$  **using** *scc-non-empty'* **by** *auto*  
**then obtain**  $\varphi_y$  **where** *\varphi\_y-in-y*:  $\varphi_y \in y$  **by** *auto*

**assume** *\varphi\_b-elem*:  $\varphi_b \in z$   
**assume**  $\bigwedge \varphi_b. \varphi_a \in a \implies \varphi_b \in y \implies (\varphi_a, \varphi_b) \in (\text{induced-phi-graph } g \ P)^*$   
**with** *assms(2) \varphi\_y-in-y*  
**have** *\varphi\_a-to-\varphi\_y*:  $(\varphi_a, \varphi_y) \in (\text{induced-phi-graph } g \ P)^*$  **using** *condensation-edge-impl-path*  
**by** *auto*

**from** *\varphi\_b-elem \varphi\_y-in-y (y, z) \in condensation-edges g P*  
**have**  $(\varphi_y, \varphi_b) \in (\text{induced-phi-graph } g \ P)^*$  **using** *condensation-edge-impl-path* **by** *auto*  
**with** *\varphi\_a-to-\varphi\_y*  
**show**  $(\varphi_a, \varphi_b) \in (\text{induced-phi-graph } g \ P)^*$  **by** *auto*

**qed** (*auto intro:condensation-edge-impl-path*)

**lemma** *condensation-acyclic: acyclic (condensation-edges g P)*

**proof** (*rule acyclicI, rule allI, rule ccontr, simp*)

**fix** *x*

Assume there is a cycle in the condensation graph.

**assume** *cyclic: (x, x) ∈ (condensation-edges g P)<sup>+</sup>*

**have** *nonrefl: (x, x) ∉ (condensation-edges g P)* **unfolding** *condensation-edges-def* **by** *auto*

Then there must be a second SCC *b* on this path.

**from** *this cyclic*

**obtain** *b* **where** *b-on-path: (x, b) ∈ (condensation-edges g P) (b, x) ∈ (condensation-edges g P)<sup>+</sup>*

**by** (*meson converse-tranclE*)

**hence** *x ∈ (condensation-nodes g P) b ∈ (condensation-nodes g P)* **using** *condensation-nodes-edges* **by** *auto*

**hence** *nodes-are-scc: is-scc (induced-phi-graph g P) x is-scc (induced-phi-graph g P) b*

**using** *scc-of-is-scc* **unfolding** *induced-phi-graph-def condensation-nodes-def* **by** *auto*

However, the existence of this path means all nodes in *b* and *x* are mutually reachable g.

**have**  $\exists \varphi_x. \varphi_x \in x \exists \varphi_b. \varphi_b \in b$  **using** *nodes-are-scc scc-non-empty' ex-in-conv* **by** *auto*

**then obtain**  $\varphi_x \varphi_b$  **where** *φxb-elem: φ<sub>x</sub> ∈ x φ<sub>b</sub> ∈ b* **by** *metis*

**with** *nodes-are-scc(1) b-on-path path-in-condensation-impl-path condensation-edge-impl-path φxb-elem(2)*

**have**  $\varphi_b \in x$

**by** *– (rule is-scc-closed)*

This however means *x* and *b* must be the same SCC, which is a contradiction to the nonreflexivity of *condensation-edges*.

**with** *nodes-are-scc φxb-elem*

**have** *x = b* **using** *is-scc-unique[of induced-phi-graph g P]* **by** *simp*

**hence**  $(x, x) \in (condensation-edges g P)$  **using** *b-on-path* **by** *simp*

**with** *nonrefl*

**show** *False* **by** *simp*

**qed**

Since the condensation graph of a set is acyclic and finite, it must have a leaf.

**lemma** *Ex-condensation-leaf:*



**assumes**  $P \neq \{\}$   
**shows**  $\exists \text{leaf}. \text{leaf} \in (\text{condensation-nodes } g P) \wedge (\forall \text{scc}. (\text{leaf}, \text{scc}) \notin \text{condensation-edges } g P)$   
**proof** –  
**from** *assms* **obtain**  $x$  **where**  $x \in \text{condensation-nodes } g P$  **unfolding** *condensation-nodes-def*  
**by** *auto*  
**show** *?thesis*  
**proof** (*rule wfE-min*)  
**from** *condensation-finite condensation-acyclic*  
**show**  $\text{wf } ((\text{condensation-edges } g P)^{-1})$  **by** (*rule finite-acyclic-wf-converse*)  
**next**  
**fix** *leaf*  
**assume** *leaf-node*:  $\text{leaf} \in \text{condensation-nodes } g P$   
**moreover**  
**assume** *leaf-is-leaf*:  $\text{scc} \notin \text{condensation-nodes } g P$  **if**  $(\text{scc}, \text{leaf}) \in (\text{condensation-edges } g P)^{-1}$  **for** *scc*  
**ultimately**  
**have**  $\text{leaf} \in \text{condensation-nodes } g P \wedge (\forall \text{scc}. (\text{leaf}, \text{scc}) \notin \text{condensation-edges } g P)$   
**using** *condensation-nodes-edges* **by** *blast*  
**thus**  $\exists \text{leaf}. \text{leaf} \in \text{condensation-nodes } g P \wedge (\forall \text{scc}. (\text{leaf}, \text{scc}) \notin \text{condensation-edges } g P)$  **by** *blast*  
**qed** *fact*  
**qed**

**lemma** *scc-in-P*:

**assumes**  $\text{scc} \in \text{condensation-nodes } g P$

**shows**  $\text{scc} \subseteq P$

**proof** –

**have**  $\text{scc} \subseteq P$  **if** *y-props*:  $\text{scc} = \text{scc-of } (\text{induced-phi-graph } g P) n$   $n \in P$  **for**  $n$

**proof** –

**from** *y-props*

**show**  $\text{scc} \subseteq P$

**proof** (*clarsimp simp:y-props(1); case-tac n = x*)

**fix**  $x$

**assume** *different*:  $n \neq x$

**assume**  $x \in \text{scc-of } (\text{induced-phi-graph } g P) n$

**hence**  $(n, x) \in (\text{induced-phi-graph } g P)^*$  **by** (*metis is-scc-connected scc-of-is-scc node-in-scc-of-node*)

**with** *different*

**have**  $(n, x) \in (\text{induced-phi-graph } g P)^+$  **by** (*metis rtranclD*)

**then obtain**  $z$  **where**  $\text{step}: (z, x) \in (\text{induced-phi-graph } g P)$  **by** (*meson tranclE*)

**from** *step*

**show**  $x \in P$  **unfolding** *induced-phi-graph-def* **by** *auto*

**qed** *simp*

**qed**

**from** *this* *assms*(1) **have**  $x \in P$  **if** *x-node*:  $x \in scc$  **for**  $x$   
**apply** –  
**apply** (*rule imageE*[*of scc scc-of (induced-phi-graph g P)*])  
**using** *condensation-nodes-def x-node* **by** *blast+*  
**thus** *?thesis* **by** *clarify*  
**qed**

**lemma** *redundant-scc-phis*:  
**assumes** *redundant-set g P scc*  $\in$  *condensation-nodes g P*  $x \in scc$   
**shows** *phi g x*  $\neq$  *None*  
**using** *assms* **by** (*meson domIff redundant-set-def scc-in-P subsetCE*)

The following lemma will be important for the main proof of this section. If  $P$  is redundant, a leaf in the condensation graph induced by  $P$  corresponds to a strongly connected set with at most one argument, thus a redundant strongly connected set exists.

Lemma 1. Every redundant set contains a redundant SCC.

**lemma** *1*:  
**assumes** *redundant-set g P*  
**shows**  $\exists scc \subseteq P$ . *redundant-scc g P scc*  
**proof** –  
**from** *assms Ex-condensation-leaf*[*of P g*]  
**obtain** *leaf* **where** *leaf-props*:  $leaf \in (condensation-nodes\ g\ P) \forall scc. (leaf, scc) \notin condensation-edges\ g\ P$   
**unfolding** *redundant-set-def* **by** *auto*  
**hence** *is-scc (induced-phi-graph g P) leaf* **unfolding** *condensation-nodes-def* **by** *auto*  
**moreover**  
**hence**  $leaf \neq \{\}$  **by** (*rule scc-non-empty'*)  
**moreover**  
**have**  $leaf \subseteq dom\ (phi\ g)$   
**apply** (*subst subset-eq, rule ballI*)  
**using** *redundant-scc-phis leaf-props*(1) *assms*(1) **by** *auto*  
**moreover**  
**from** *assms*  
**obtain** *pred* **where** *pred-props*:  $pred \in allVars\ g \forall \varphi \in P. \forall \varphi'. phiArg\ g\ \varphi\ \varphi' \longrightarrow \varphi' \in P \cup \{pred\}$  **unfolding** *redundant-set-def* **by** *auto*  
**{**

Any argument of a  $\phi$ -function in the leaf SCC which is *not* in the leaf SCC itself must be the unique argument of  $P$

**fix**  $\varphi\ \varphi'$

**consider** (*in-P*)  $\varphi' \notin leaf \wedge \varphi' \in P \mid (neither)\ \varphi' \notin leaf \wedge \varphi' \notin P \cup \{pred\} \mid \varphi' \notin leaf \wedge \varphi' \in \{pred\} \mid \varphi' \in leaf$  **by** *auto*

**hence**  $\varphi' \in \text{leaf} \cup \{\text{pred}\}$  **if**  $\varphi \in \text{leaf}$  **and**  $\text{phiArg } g \varphi \varphi'$   
**proof cases**  
**case** *in-P* — In this case *leaf* wasn't really a leaf, a contradiction  
**moreover**  
**from** *in-P* **that** *leaf-props(1) scc-in-P[of leaf g P]*  
**have**  $(\varphi, \varphi') \in \text{induced-phi-graph } g P$  **unfolding** *induced-phi-graph-def* **by** *auto*  
**ultimately**  
**have**  $(\text{leaf}, \text{scc-of } (\text{induced-phi-graph } g P) \varphi') \in \text{condensation-edges } g P$  **unfolding**  
*condensation-edges-def*  
**using** *leaf-props(1) that is-scc (induced-phi-graph g P) leaf*  
**apply** —  
**apply** *clarsimp*  
**apply** *(rule conjI)*  
**prefer** 2  
**apply** *auto[1]*  
**unfolding** *condensation-nodes-def*  
**by** *(metis (no-types, lifting) is-scc-unique node-in-scc-of-node pair-imageI scc-of-is-scc)*  
**with** *leaf-props(2)*  
**show** *?thesis* **by** *auto*  
**next**  
**case** *neither* — In which case *P* itself wasn't redundant, a contradiction  
**with** *that leaf-props pred-props*  
**have**  $\neg \text{redundant-set } g P$  **unfolding** *redundant-set-def*  
**by** *(meson rev-subsetD scc-in-P)*  
**with** *assms*  
**show** *?thesis* **by** *auto*  
**qed** *auto* — the other cases are trivial  
**}**  
**with** *pred-props(1)*  
**have**  $\exists v' \in \text{allVars } g. \forall \varphi \in \text{leaf}. \forall \varphi'. \text{phiArg } g \varphi \varphi' \longrightarrow \varphi' \in \text{leaf} \cup \{v'\}$  **by** *auto*  
**ultimately**  
**have** *redundant-scc g P leaf* **unfolding** *redundant-scc-def redundant-set-def* **by** *auto*  
**thus** *?thesis* **using** *leaf-props(1) scc-in-P* **by** *blast*  
**qed**

## Proof of Minimality

We inductively define the reachable g-set of a  $\phi$ -function as all  $\phi$ -functions reachable g from a given node via an unbroken chain of  $\phi$  argument edges to unnecessary  $\phi$ -functions.

**inductive-set** *reachable* ::  $'g \Rightarrow 'val \Rightarrow 'val \text{ set}$   
**for**  $g :: 'g$  **and**  $\varphi :: 'val$   
**where** *refl: unnecessaryPhi g  $\varphi \Longrightarrow \varphi \in \text{reachable } g \varphi$*   
| *step:  $\varphi' \in \text{reachable } g \varphi \Longrightarrow \text{phiArg } g \varphi' \varphi'' \Longrightarrow \text{unnecessaryPhi } g \varphi'' \Longrightarrow \varphi'' \in \text{reachable } g \varphi$*

**lemma** *reachable-props*:

**assumes**  $\varphi' \in \text{reachable } g \ \varphi$   
**shows**  $(\text{phiArg } g)^{**} \ \varphi \ \varphi'$  **and** *unnecessaryPhi*  $g \ \varphi'$   
**using** *assms*  
**by** (*induction*  $\varphi'$  *rule: reachable.induct*) *auto*

We call the transitive arguments of a  $\phi$ -function not in its reachable g-set the "true arguments" of this  $\phi$ -function.

**definition** [*simp*]:  $\text{trueArgs } g \ \varphi \equiv \{\varphi'. \ \varphi' \notin \text{reachable } g \ \varphi\} \cap \{\varphi'. \ \exists \varphi'' \in \text{reachable } g \ \varphi. \ \text{phiArg } g \ \varphi'' \ \varphi'\}$

**lemma** *preds-finite*: *finite* ( $\text{trueArgs } g \ \varphi$ )

**proof** (*rule ccontr*)

**assume** *infinite* ( $\text{trueArgs } g \ \varphi$ )

**hence** *a*: *infinite*  $\{\varphi'. \ \exists \varphi'' \in \text{reachable } g \ \varphi. \ \text{phiArg } g \ \varphi'' \ \varphi'\}$  **by** *auto*

**have** *phiarg-set*:  $\{\varphi'. \ \exists \varphi. \ \text{phiArg } g \ \varphi \ \varphi'\} = \bigcup (\text{set } \{b. \ \exists a. \ \text{phi } g \ a = \text{Some } b\})$  **unfolding** *phiArg-def* **by** *auto*

If the true arguments of a  $\phi$ -function are infinite in number, there must be an infinite number of  $\phi$ -functions. . .

**have** *infinite*  $\{\varphi'. \ \exists \varphi. \ \text{phiArg } g \ \varphi \ \varphi'\}$

**by** (*rule infinite-super*[*of*  $\{\varphi'. \ \exists \varphi'' \in \text{reachable } g \ \varphi. \ \text{phiArg } g \ \varphi'' \ \varphi'\}$ ]) (*auto simp*: *a*)

**with** *phiarg-set*

**have** *infinite* (*ran* ( $\text{phi } g$ )) **unfolding** *ran-def phiArg-def* **by** *clarsimp*

Which cannot be.

**thus** *False* **by** (*simp add: phi-finite map-dom-ran-finite*)

**qed**

Any unnecessary  $\phi$  with less than 2 true arguments induces with  $\text{reachable } g \ \varphi$  a redundant set itself.

**lemma** *few-preds-redundant*:

**assumes**  $\text{card } (\text{trueArgs } g \ \varphi) < 2$  *unnecessaryPhi*  $g \ \varphi$

**shows** *redundant-set*  $g \ (\text{reachable } g \ \varphi)$

**unfolding** *redundant-set-def*

**proof** (*intro conjI*)

**from** *assms*

**show**  $\text{reachable } g \ \varphi \neq \{\}$

**using** *empty-iff reachable.intros(1)* **by** *auto*

**next**

```

from assms(2)
show reachable g  $\varphi \subseteq \text{dom} (\text{phi } g)$ 
  by (metis domIff reachable.cases subsetI unnecessaryPhi-def)
next
  from assms(1)
  consider (single) card (trueArgs g  $\varphi) = 1 \mid (\text{empty}) \text{card} (trueArgs g  $\varphi) = 0$  by force$ 
  thus  $\exists \text{pred} \in \text{allVars } g. \forall \varphi' \in \text{reachable } g \varphi. \forall \varphi''. \text{phiArg } g \varphi' \varphi'' \longrightarrow \varphi'' \in \text{reachable } g \varphi$ 
   $\cup \{\text{pred}\}$ 
  proof cases
    case single
    then obtain pred where pred-prop: trueArgs g  $\varphi = \{\text{pred}\}$  using card-eq-1-singleton
  by blast
  hence pred  $\in$  allVars g by (auto intro: Int-Collect phiArg-in-allVars)
  moreover
  from pred-prop
  have  $\forall \varphi' \in \text{reachable } g \varphi. \forall \varphi''. \text{phiArg } g \varphi' \varphi'' \longrightarrow \varphi'' \in \text{reachable } g \varphi \cup \{\text{pred}\}$  by
auto
  ultimately
  show ?thesis by auto
  next
  case empty
  from allDefs-in-allVars[of - g defNode g  $\varphi$ ] assms
  have phi-var:  $\varphi \in \text{allVars } g$  unfolding unnecessaryPhi-def phiDefs-def allDefs-def
defNode-def phi-def trueArgs-def
  by (clarsimp simp: domIff phis-in- $\alpha n$ )
  from empty assms(1)
  have no-preds: trueArgs g  $\varphi = \{\}$  by (subst card-0-eq[OF preds-finite, symmetric])
auto
  show ?thesis
  proof (rule bexI, rule ballI, rule allI, rule impI)
  fix  $\varphi' \varphi''$ 
  assume phis-props:  $\varphi' \in \text{reachable } g \varphi$  phiArg g  $\varphi' \varphi''$ 
  with no-preds
  have  $\varphi'' \in \text{reachable } g \varphi$ 
  unfolding trueArgs-def
  proof –
  from phis-props
  have  $\varphi'' \in \{\varphi'. \exists \varphi'' \in \text{reachable } g \varphi. \text{phiArg } g \varphi'' \varphi'\}$  by auto
  with phis-props no-preds
  show  $\varphi'' \in \text{reachable } g \varphi$  unfolding trueArgs-def by auto
  qed
  thus  $\varphi'' \in \text{reachable } g \varphi \cup \{\varphi\}$  by simp
  qed (auto simp: phi-var)
qed
qed

```

```

lemma phiArg-trancl-same-var:
assumes  $(\text{phiArg } g)^{++} \varphi \ n$ 
shows  $\text{var } g \ \varphi = \text{var } g \ n$ 
using assms
apply (induction rule: tranclp-induct)
  apply (rule phiArg-same-var[symmetric])
  apply simp
using phiArg-same-var by auto

```

The following path extension lemma will be used a number of times in the inner induction of the main proof. Basically, the idea is to extend a path ending in a  $\phi$  argument to the corresponding  $\phi$ -functionwhile preserving disjointness to a second path.

```

lemma phiArg-disjoint-paths-extend:
assumes  $\text{var } g \ r = V$  and  $\text{var } g \ s = V$  and  $r \in \text{allVars } g$  and  $s \in \text{allVars } g$ 
and  $V \in \text{oldDefs } g \ n$  and  $V \in \text{oldDefs } g \ m$ 
and  $g \vdash n - ns \rightarrow \text{defNode } g \ r$  and  $g \vdash m - ms \rightarrow \text{defNode } g \ s$ 
and  $\text{set } ns \cap \text{set } ms = \{\}$ 
and phiArg  $g \ \varphi_r \ r$ 
obtains  $ns'$ 
where  $g \vdash n - ns @ ns' \rightarrow \text{defNode } g \ \varphi_r$ 
and  $\text{set } (\text{butlast } (ns @ ns')) \cap \text{set } ms = \{\}$ 
proof (cases  $r = \varphi_r$ )
  case (True)

```

If the node to extend the path to is already the endpoint, the lemma is trivial.

```

  with assms(7,8,9) in-set-butlastD
  have  $g \vdash n - ns @ [] \rightarrow \text{defNode } g \ \varphi_r$   $\text{set } (\text{butlast } (ns @ [])) \cap \text{set } ms = \{\}$ 
    by simp-all fastforce
  with that show ?thesis .
next
  case False

```

It suffices to obtain any path from  $r$  to  $\varphi_r$ . However, since we'll need the corresponding predecessor of  $\varphi_r$  later, we must do this as follows:

```

  from assms(10)
  have  $\varphi_r \in \text{allVars } g$  unfolding phiArg-def
    by (metis allDefs-in-allVars phiDefs-in-allDefs phi-def phi-phiDefs phis-in- $\alpha n$ )
  with assms(10)
  obtain  $rs' \ \text{pred}_{\varphi_r}$  where  $rs' \text{-props}: g \vdash \text{defNode } g \ r - rs' \rightarrow \text{pred}_{\varphi_r}$  old.EntryPath  $g \ rs' \ r$ 
     $\in \text{phiUses } g \ \text{pred}_{\varphi_r}$   $\text{pred}_{\varphi_r} \in \text{set } (\text{old.predecessors } g \ (\text{defNode } g \ \varphi_r))$ 
    by (rule phiArg-path-ex')

  def  $rs \equiv rs' @ [\text{defNode } g \ \varphi_r]$ 

```

```

from rs'-props(2,1) old.EntryPath-distinct old.path2-hd
have rs'-loopfree: defNode g r ∉ set (tl rs') by (simp add: Misc.distinct-hd-tl)

from False assms have defNode g φr ≠ defNode g r
apply –
apply (rule phiArg-distinct-nodes)
apply (auto intro:phiArg-in-allVars)
unfolding phiArg-def by (metis allDefs-in-allVars phiDefs-in-allDefs phi-def phi-phiDefs
phis-in-αn)

from rs'-props
have rs-props: g ⊢ defNode g r – rs → defNode g φr length rs > 1 defNode g r ∉ set (tl
rs)
apply (subgoal-tac defNode g r = hd rs')
prefer 2 using rs'-props(1)
apply (rule old.path2-hd)
using old.path2-snoc old.path2-def rs'-props(1) rs-def rs'-loopfree ⟨defNode g φr ≠
defNode g r⟩ by auto

show thesis
proof (cases set (butlast rs) ∩ set ms = {})
case inter-empty: True

```

If the intersection of these is empty, *tl rs* is already the extension we're looking for

```

show thesis
proof (rule that)
show set (butlast (ns @ tl rs)) ∩ set ms = {}
proof (rule ccontr, simp only: ex-in-conv[symmetric])
assume  $\exists x. x \in \text{set } (\text{butlast } (ns @ tl rs)) \cap \text{set } ms$ 
then obtain x where x-props: x ∈ set (butlast (ns @ tl rs)) x ∈ set ms by auto
with rs-props(2)
consider (in-ns) x ∈ set ns | (in-rs) x ∈ set (butlast (tl rs)) by (metis Un-iff
butlast-append in-set-butlastD set-append)
thus False
apply (cases)
using x-props(2) assms(9)
apply (simp add: disjoint-elem)
by (metis x-props(2) inter-empty in-set-tlD List.butlast-tl disjoint-iff-not-equal)
qed
qed (auto intro:assms(7) rs-props(1) old.path2-app)
next
case inter-ex: False

```

If the intersection is nonempty, there must be a first point of intersection *i*.

```

from inter-ex assms(7,8) rs-props

```

```

obtain  $i$   $ri$  where  $ri$ -props:  $g \vdash \text{defNode } g \ r - ri \rightarrow i \ i \in \text{set } ms \ \forall n \in \text{set } (\text{butlast } ri). \ n \notin \text{set } ms \ \text{prefixeq } ri \ rs$ 
apply  $-$ 
apply ( $rule \ \text{old.path2-split-first-prop}$ [of  $g \ \text{defNode } g \ r \ rs \ \text{defNode } g \ \varphi_r$ , where  $P = \lambda m. m \in \text{set } ms$ ])
apply  $\text{blast}$ 
apply ( $metis \ \text{disjoint-iff-not-equal in-set-butlastD}$ )
by  $\text{blast}$ 
with  $\text{assms}(8) \ \text{old.path2-prefix-ex}$ 
obtain  $ms'$  where  $ms'$ -props:  $g \vdash m - ms' \rightarrow i \ \text{prefixeq } ms' \ ms \ i \notin \text{set } (\text{butlast } ms')$ 
by  $\text{blast}$ 

```

We proceed by case distinction:

- if  $i = \text{defNode } g \ \varphi_r$ , the path  $ri$  is already the path extension we're looking for
- Otherwise, the fact that  $i$  is on the path from  $\phi$  argument to the  $\phi$  itself leads to a contradiction. However, we still need to distinguish the cases of whether  $m = i$

```

consider ( $ri$ -is-valid)  $i = \text{defNode } g \ \varphi_r \mid (m$ -i-same)  $i \neq \text{defNode } g \ \varphi_r \ m = i \mid (m$ -i-differ)  $i \neq \text{defNode } g \ \varphi_r \ m \neq i$  by  $auto$ 

```

```

thus  $\text{thesis}$ 
proof ( $\text{cases}$ )
case  $ri$ -is-valid

```

$ri$  is a valid path extension.

```

with  $\text{assms}(7) \ ri$ -props(1)
have  $g \vdash n - ns @ (tl \ ri) \rightarrow \text{defNode } g \ \varphi_r$  by  $auto$ 

```

**moreover**

```

have  $\text{set } (\text{butlast } (ns @ (tl \ ri))) \cap \text{set } ms = \{\}$ 

```

```

proof ( $rule \ \text{ccontr}$ )

```

```

assume  $\text{contr}: \text{set } (\text{butlast } (ns @ tl \ ri)) \cap \text{set } ms \neq \{\}$ 

```

```

from  $\text{this}$ 

```

```

obtain  $x$  where  $x$ -props:  $x \in \text{set } (\text{butlast } (ns @ tl \ ri)) \ x \in \text{set } ms$  by  $auto$ 

```

```

with  $\text{assms}(9)$  have  $x \notin \text{set } ns$  by  $auto$ 

```

```

with  $x$ -props  $\langle g \vdash n - ns @ tl \ ri \rightarrow \text{defNode } g \ \varphi_r \rangle \langle \text{defNode } g \ \varphi_r \neq \text{defNode } g \ r \rangle$ 

```

```

 $\text{assms}(7)$ 

```

```

have  $x \in \text{set } (\text{butlast } (tl \ ri))$ 

```

```

by ( $metis \ \text{Un-iff append-Nil2 butlast-append old.path2-last set-append}$ )

```

```

with  $x$ -props(2)  $ri$ -props(3)

```

```

show  $\text{False}$  by ( $metis \ \text{FormalSSA-Misc.in-set-tlD List.butlast-tl}$ )

```

```

qed

```

```

ultimately

```



**show thesis by** (rule that)  
**next**  
**case** *m-i-same*

If  $m = i$ , we have, with  $m$ , a variable definition on the path from a  $\phi$ -function to its argument. This constitutes a contradiction to the conventional property.

**note** *rs'-props(1) rs'-loopfree*  
**moreover have**  $r \in allDefs\ g\ (defNode\ g\ r)$  **by** (*simp add: assms(3)*)  
**moreover from** *rs'-props(3)* **have**  $r \in allUses\ g\ pred_{\varphi_r}$  **unfolding** *allUses-def* **by** *simp*

**moreover**  
**from** *rs-props(1) m-i-same rs-def ri-props(1,2,4)*  $\langle defNode\ g\ \varphi_r \neq defNode\ g\ r \rangle$   
*assms(7,9)*  
**have**  $m \in set\ (tl\ rs')$   
**by** (*metis disjoint-elem hd-append in-hd-or-tl-conv in-prefix list.sel(1) old.path2-hd old.path2-last old.path2-last-in-ns prefix-req-snoc*)

**moreover**  
**from** *assms(6)* **obtain**  $def_m$  **where**  $def_m \in allDefs\ g\ m\ var\ g\ def_m = V$  **unfolding** *oldDefs-def* **using** *defs-in-allDefs* **by** *blast*

**ultimately**  
**have**  $var\ g\ def_m \neq var\ g\ r$  **by**  $-$  (*rule conventional, simp-all*)  
**with**  $\langle var\ g\ def_m = V \rangle$  *assms(1)*  
**have** *False* **by** *simp*  
**thus** *?thesis* **by** *simp*

**next**  
**case** *m-i-differ*

If  $m \neq i$ ,  $i$  constitutes a proper path convergence point.

**have** *old.pathsConverge g m ms' n (ns @ tl ri) i*  
**proof** (*rule old.pathsConvergeI*)  
**show**  $1 < length\ ms'$  **using** *m-i-differ ms'-props old.path2-nontriv* **by** *blast*  
**next**  
**show**  $1 < length\ (ns\ @\ tl\ ri)$   
**using** *ri-props old.path2-nontriv assms(9)* **by** (*metis assms(7) disjoint-elem old.path2-app old.path2-hd-in-ns*)  
**next**  
**show**  $set\ (butlast\ ms') \cap set\ (butlast\ (ns\ @\ tl\ ri)) = \{\}$   
**proof** (*rule ccontr*)  
**assume**  $set\ (butlast\ ms') \cap set\ (butlast\ (ns\ @\ tl\ ri)) \neq \{\}$   
**then obtain**  $i'$  **where**  $i' \in set\ (butlast\ ms')$   $i' \in set\ (butlast\ (ns\ @\ tl\ ri))$  **by** *auto*

```

with ms'-props(2)
have i'-not-in-ms:  $i' \in \text{set } (\text{butlast } ms)$  by (metis in-set-butlast-appendI prefixeqE)

with assms(9)
show False
proof (cases i' \notin set ns)
  case True
    with i'-props(2)
    have  $i' \in \text{set } (\text{butlast } (tl \ ri))$ 
      by (metis Un-iff butlast-append in-set-butlastD set-append)
    hence  $i' \in \text{set } (\text{butlast } ri)$  by (simp add:in-set-tlD List.butlast-tl)
    with i'-not-in-ms ri-props(3)
    show False by (auto dest:in-set-butlastD)
    qed (meson disjoint-elem in-set-butlastD)
  qed
qed (auto intro: assms(7) ri-props(1) old.path2-app ms'-props(1))

```

At this intersection of paths we can find a  $\phi$ -function.

```

from this assms(6,5)
have necessaryPhi  $g \ V \ i$  by (rule necessaryPhiI)

```

Before we can conclude that there is indeed a  $\phi$  at  $i$ , we have to prove a couple of technicalities...

```

moreover
from m-i-differ ri-props(1,4) rs-def old.path2-last prefixeq-snoc
have ri-rs'-prefix: prefixeq ri rs' by fastforce
  then obtain rs'-rest where rs'-rest-prop:  $rs' = ri @ rs'\text{-rest}$  using prefixeqE by
auto
from old.path2-last[OF ri-props(1)] last-snoc[of - i] obtain tmp where  $ri = tmp @ [i]$ 
apply (subgoal-tac ri \neq [])
prefer 2
using ri-props(1) apply (simp add: old.path2-not-Nil)
apply (rule-tac that)
using append-butlast-last-id[symmetric] by auto
with rs'-rest-prop have rs'-rest-def:  $rs' = tmp @ i \# rs'\text{-rest}$  by auto
with rs'-props(1) have  $g \vdash i - i \# rs'\text{-rest} \rightarrow \text{pred}_{\varphi r}$ 
by (simp add:old.path2-split)
moreover
note  $\langle \text{var } g \ r = V \rangle$  [simp]
from rs'-props(3)
have  $r \in \text{allUses } g \ \text{pred}_{\varphi r}$  unfolding allUses-def by simp

moreover
from  $\langle \text{defNode } g \ r \notin \text{set } (tl \ rs') \rangle$  rs'-rest-def
have defNode  $g \ r \notin \text{set } rs'\text{-rest}$  by auto

```

```

with  $\langle g \vdash i - i\#rs'-rest \rightarrow pred_{\varphi r} \rangle$ 
have  $\bigwedge x. x \in set\ rs'-rest \implies r \notin allDefs\ g\ x$ 
by (metis defNode-eq list.distinct(1) list.sel(3) list.set-cases old.path2-cases old.path2-in- $\alpha n$ )

moreover
from assms(7,9)  $\langle g \vdash i - i\#rs'-rest \rightarrow pred_{\varphi r} \rangle$  ri-props(2)
have  $r \notin defs\ g\ i$ 
by (metis defNode-eq defs-in-allDefs disjoint-elem old.path2-hd-in- $\alpha n$  old.path2-last-in-ns)
ultimately

```

The convergence property gives us that there is a  $\phi$  in the last node fulfilling *necessaryPhi* on a path to a use of  $r$  without a definition of  $r$ . Thus  $i$  bears a  $\phi$ -function for the value of  $r$ .

```

have  $\exists y. phis\ g\ (i, r) = Some\ y$ 
by (rule convergence-prop [where  $g=g$  and  $n=i$  and  $v=r$  and  $ns=i\#rs'-rest$ ,
simplified])
moreover

from  $\langle g \vdash n - ns \rightarrow defNode\ g\ r \rangle$  have  $defNode\ g\ r \in set\ ns$  by auto
with  $\langle set\ ns \cap set\ ms = \{\} \rangle \langle i \in set\ ms \rangle$  have  $i \neq defNode\ g\ r$  by auto
moreover

from ms'-props(1) have  $i \in set\ (\alpha n\ g)$  by auto
moreover

have  $defNode\ g\ r \in set\ (\alpha n\ g)$  by (simp add: assms(3))

```

However, we now have two definitions of  $r$ : one in  $i$ , and one in  $defNode\ g\ r$ , which we know to be distinct. This is a contradiction to the *allDefs-disjoint*-property.

```

ultimately have False
using allDefs-disjoint [where  $g=g$  and  $n=i$  and  $m=defNode\ g\ r$ ]
unfolding allDefs-def phiDefs-def
apply clarsimp
apply (erule-tac c=r in equalityCE)
using phi-def phis-phi by auto
thus ?thesis by simp
qed
qed
qed

```

```

lemma reachable-same-var:
assumes  $\varphi' \in reachable\ g\ \varphi$ 
shows  $var\ g\ \varphi = var\ g\ \varphi'$ 
using assms by (metis Nitpick.rtranclp-unfold phiArg-trancl-same-var reachable-props(1))

```

**lemma**  $\varphi$ -node-no-defs:  
**assumes** *unnecessaryPhi*  $g \ \varphi \ \varphi \in \text{allVars } g \ \text{var } g \ \varphi \in \text{oldDefs } g \ n$   
**shows**  $\text{defNode } g \ \varphi \neq n$   
**using** *assms simpleDefs-phiDefs-var-disjoint defNode(1) not-None-eq phi-phiDefs*  
**unfolding** *unnecessaryPhi-def* **by** *auto*

**lemma** *defNode-differ-aux*:  
**assumes**  $\varphi_s \in \text{reachable } g \ \varphi \ \varphi \in \text{allVars } g \ s \in \text{allVars } g \ \varphi_s \neq s \ \text{var } g \ \varphi = \text{var } g \ s$   
**shows**  $\text{defNode } g \ \varphi_s \neq \text{defNode } g \ s$  **unfolding** *reachable-def*  
**proof** (*rule ccontr*)  
**assume**  $\neg \text{defNode } g \ \varphi_s \neq \text{defNode } g \ s$   
**hence** *eq*:  $\text{defNode } g \ \varphi_s = \text{defNode } g \ s$  **by** *simp*  
**from** *assms(1)*  
**have** *vars-eq*:  $\text{var } g \ \varphi = \text{var } g \ \varphi_s$   
**apply**  $-$   
**apply** (*cases*  $\varphi = \varphi_s$ )  
**apply** *simp*  
**apply** (*rule phiArg-trancl-same-var*)  
**apply** (*drule reachable-props*)  
**unfolding** *reachable-def* **by** (*meson IntD1 mem-Collect-eq rtranclpD*)  
**have**  $\varphi_s$ -in-allVars:  $\varphi_s \in \text{allVars } g$  **unfolding** *reachable-def*  
**proof** (*cases*  $\varphi = \varphi_s$ )  
**case** *False*  
**with** *assms(1)*  
**obtain**  $\varphi'$  **where** *phiArg*  $g \ \varphi' \ \varphi_s$  **by** (*metis rtranclp.cases reachable-props(1)*)  
**thus**  $\varphi_s \in \text{allVars } g$  **by** (*rule phiArg-in-allVars*)  
**next**  
**case** *eq*: *True*  
**with** *assms(2)*  
**show**  $\varphi_s \in \text{allVars } g$  **by** (*subst eq[symmetric]*)  
**qed**  
**from** *eq*  $\varphi_s$ -in-allVars *assms(3,4)*  
**have**  $\text{var } g \ \varphi_s \neq \text{var } g \ s$  **by**  $-$  (*rule defNode-var-disjoint*)  
**with** *vars-eq* *assms(5)*  
**show** *False* **by** *auto*  
**qed**

Theorem 1. A graph which does not contain any redundant SCCs is minimal according to Cytron et al.'s definition of minimality.

**theorem** *no-redundant-SCC-minimal*:  
**assumes**  $\neg(\exists P \text{ scc. redundant-scc } g \ P \ \text{scc})$   
**shows** *cytronMinimal*  $g$

**proof** (*rule ccontr*)

Assume there are no redundant SCCs. Thus with lemma 1, there are no redundant sets.

```

from assms 1
have no-redundant-set:  $\neg(\exists s. \text{redundant-set } g \ s)$  by blast

assume  $\neg$ cytronMinimal g

```

Assume the graph is not Cytron-minimal. Thus there is a  $\phi$ -function which does not sit at the convergence point of multiple liveness intervals.

```

then obtain  $\varphi$  where  $\varphi$ -props: unnecessaryPhi g  $\varphi$   $\varphi \in \text{allVars } g$   $\varphi \in \text{reachable } g$ 
using cytronMinimal-def unnecessaryPhi-def reachable-def unnecessaryPhi-def reachable.intros by auto

```

We consider the reachable g-set of  $\varphi$ . If  $\varphi$  has less than two true arguments, we know it to be a redundant set, a contradiction. Otherwise, we know there to be at least two paths from different definitions leading into the reachable g set of  $\varphi$ .

```

consider (nontrivial) card (trueArgs g  $\varphi$ )  $\geq 2$  | (trivial) card (trueArgs g  $\varphi$ )  $< 2$  using
linorder-not-le by auto
thus False
proof cases
  case trivial

```

If there are less than 2 true arguments of this set, the set is trivially redundant (see *few-preds-redundant*).

```

from this  $\varphi$ -props(1)
have redundant-set g (reachable g  $\varphi$ ) by (rule few-preds-redundant)
with no-redundant-set
show False by simp
next
  case nontrivial

```

If there are two or more necessary arguments, there must be disjoint paths from Defs to two of these  $\phi$ -functions.

```

then obtain r s  $\varphi_r \varphi_s$  where assign-nodes-props:
  r  $\neq$  s  $\varphi_r \in \text{reachable } g$   $\varphi_s \in \text{reachable } g$ 
   $\neg$  unnecessaryPhi g r  $\neg$  unnecessaryPhi g s
  r  $\in$  {n. (phiArg g)**  $\varphi$  n} s  $\in$  {n. (phiArg g)**  $\varphi$  n}
  phiArg g  $\varphi_r$  r phiArg g  $\varphi_s$  s
apply simp
apply (rule set-take-two[OF nontrivial])
apply simp

```

```

by (meson reachable.intros(2) reachable-props(1) rtranclp-tranclp-tranclp tranclp.r-into-trancl
tranclp-into-rtranclp)
moreover from assign-nodes-props
have  $\varphi$ -r-s-uneq:  $\varphi \neq r \varphi \neq s$  using  $\varphi$ -props by auto
moreover
from assign-nodes-props this
have r-s-in-tranclp:  $(\text{phiArg } g)^{++} \varphi r (\text{phiArg } g)^{++} \varphi s$ 
by (meson mem-Collect-eq rtranclpD) (meson assign-nodes-props(7)  $\varphi$ -r-s-uneq(2)
mem-Collect-eq rtranclpD)
from this
obtain  $V$  where  $V$ -props:  $\text{var } g r = V \text{ var } g s = V \text{ var } g \varphi = V$  by (metis
phiArg-tranclp-same-var)
moreover
from r-s-in-tranclp
have r-s-allVars:  $r \in \text{allVars } g s \in \text{allVars } g$  by (metis phiArg-in-allVars tran
clp.cases)+
moreover
from  $V$ -props defNode-var-disjoint r-s-allVars assign-nodes-props(1)
have r-s-defNode-distinct:  $\text{defNode } g r \neq \text{defNode } g s$  by auto
ultimately
obtain  $n ns m ms$  where r-s-path-props:  $V \in \text{oldDefs } g n g \vdash n - ns \rightarrow \text{defNode } g r V$ 
 $\in \text{oldDefs } g m g \vdash m - ms \rightarrow \text{defNode } g s$ 
 $\text{set } ns \cap \text{set } ms = \{\}$  by (auto intro: unnecessaryPhis-disjoint-paths[of  $g r s$ ])

have n-m-distinct:  $n \neq m$ 
proof (rule ccontr)
assume n-m:  $\neg n \neq m$ 
with r-s-path-props(2) old.path2-hd-in-ns
have  $n \in \text{set } ns$  by blast
moreover
from n-m r-s-path-props(4) old.path2-hd-in-ns
have  $n \in \text{set } ms$  by blast
ultimately
show False using r-s-path-props(5) by auto
qed

```

These paths can be extended into paths reaching  $\phi$ -functions in our set.

```

from  $V$ -props r-s-allVars r-s-path-props assign-nodes-props
obtain  $rs$  where rs-props:  $g \vdash n - ns @ rs \rightarrow \text{defNode } g \varphi_r \text{ set } (\text{butlast } (ns @ rs)) \cap \text{set}$ 
 $ms = \{\}$ 
using phiArg-disjoint-paths-extend by blast

```

(In fact, we can prove that  $\text{set } (ns @ rs) \cap \text{set } ms = \{\}$ , which we need for the next path extension.)

```

have  $\text{defNode } g \varphi_r \notin \text{set } ms$ 

```

```

proof (rule ccontr)
  assume  $\varphi_r$ -in-ms:  $\neg \text{defNode } g \ \varphi_r \notin \text{set } ms$ 
  from this r-s-path-props(4)
  obtain  $ms'$  where  $ms'$ -props:  $g \vdash m -ms' \rightarrow \text{defNode } g \ \varphi_r \text{ prefixeq } ms' \ ms$  by  $\neg$ (rule
old.path2-prefix-ex[of  $g \ m \ ms \ \text{defNode } g \ s \ \text{defNode } g \ \varphi_r$ ], auto)

  have old.pathsConverge  $g \ n \ (ns@rs) \ m \ ms' \ (\text{defNode } g \ \varphi_r)$ 
  proof (rule old.pathsConvergeI)
    show set (butlast (ns @ rs))  $\cap$  set (butlast  $ms'$ ) = {}
    proof (rule ccontr)
      assume set (butlast (ns @ rs))  $\cap$  set (butlast  $ms'$ )  $\neq$  {}
      then obtain  $c$  where  $c$ -props:  $c \in \text{set } (butlast \ (ns@rs)) \ c \in \text{set } (butlast \ ms')$  by
auto
      from this(2)  $ms'$ -props(2)
      have  $c \in \text{set } ms$  by (simp add: in-prefix in-set-butlastD)
      with  $c$ -props rs-props(2)
      show False by auto
    qed
  next
  have  $m$ - $n$ - $\varphi_r$ -differ:  $n \neq \text{defNode } g \ \varphi_r \ m \neq \text{defNode } g \ \varphi_r$ 
  using assign-nodes-props(2,3,4,5) V-props r-s-path-props  $\varphi_r$ -in-ms
  apply fastforce
  using V-props(1)  $\varphi_r$ -in-ms assign-nodes-props(8) old.path2-in- $\alpha n$  phiArg-def
phiArg-same-var r-s-path-props(3,4) simpleDefs-phiDefs-var-disjoint
  by auto
  with  $ms'$ -props(1)
  show  $1 < \text{length } ms'$  using old.path2-nontriv by simp
  from  $m$ - $n$ - $\varphi_r$ -differ rs-props(1)
  show  $1 < \text{length } (ns@rs)$  using old.path2-nontriv by blast
  qed (auto intro: rs-props set-mono-prefixeq  $ms'$ -props)
  with V-props r-s-path-props
  have necessaryPhi'  $g \ \varphi_r$  unfolding necessaryPhi-def using assign-nodes-props(8)
phiArg-same-var by auto
  with reachable-props(2)[OF assign-nodes-props(2)]
  show False unfolding unnecessaryPhi-def by simp
  qed

with rs-props
have aux: set  $ms \cap \text{set } (ns @ rs) = \{\}$ 
  by (metis disjoint-iff-not-equal not-in-butlast old.path2-last)

have  $\varphi_r$ -V: var  $g \ \varphi_r = V$ 
  using V-props(1) assign-nodes-props(8) phiArg-same-var by auto

have  $\varphi_r$ -allVars:  $\varphi_r \in \text{allVars } g$ 
  by (meson phiArg-def assign-nodes-props(8) allDefs-in-allVars old.path2-tl-in- $\alpha n$ 

```

*phiDefs-in-allDefs phi-phiDefs rs-props*)

```

from V-props(2)  $\varphi_r$ -V r-s-allVars(2)  $\varphi_r$ -allVars r-s-path-props(3) r-s-path-props(1)
      r-s-path-props(4) rs-props(1) aux assign-nodes-props(9)
obtain ss where ss-props:  $g \vdash m -ms@ss \rightarrow defNode\ g\ \varphi_s\ set\ (butlast\ (ms@ss)) \cap$ 
set ( $butlast\ (ns@rs) = \{\}$ )
by (rule phiArg-disjoint-paths-extend) (metis disjoint-iff-not-equal in-set-butlastD)

def  $p_m \equiv ms@ss$ 
def  $p_n \equiv ns@rs$ 

have ind-props:  $g \vdash m -p_m \rightarrow defNode\ g\ \varphi_s\ g \vdash n -p_n \rightarrow defNode\ g\ \varphi_r\ set\ (butlast$ 
 $p_m) \cap set\ (butlast\ p_n) = \{\}$ 
using rs-props(1) ss-props p_m-def p_n-def by auto

```

The following case will occur twice in the induction, with swapped identifiers, so we're proving it outside. Basically, if the paths  $p_m$  and  $p_n$  intersect, the first such intersection point must be a  $\phi$ -function in *reachable*  $g\ \varphi$ , yielding the path convergence we seek.

```

have path-crossing-yields-convergence:
   $\exists \varphi_z \in reachable\ g\ \varphi.$   $\exists ns\ ms.$  old.pathsConverge  $g\ n\ ns\ m\ ms\ (defNode\ g\ \varphi_z)$ 
if  $\varphi_r \in reachable\ g\ \varphi$  and  $\varphi_s \in reachable\ g\ \varphi$  and  $g \vdash n -p_n \rightarrow defNode\ g\ \varphi_r$ 
and  $g \vdash m -p_m \rightarrow defNode\ g\ \varphi_s$  and  $set\ (butlast\ p_m) \cap set\ (butlast\ p_n) = \{\}$ 
and  $set\ p_m \cap set\ p_n \neq \{\}$ 
for  $\varphi_r\ \varphi_s\ p_m\ p_n$ 
proof -
from that(6) split-list-first-propE
obtain  $p_m1\ n_z\ p_m2$  where n_z-props:  $n_z \in set\ p_n\ p_m = p_m1 @ n_z \# p_m2\ \forall n \in set$ 
 $p_m1.$   $n \notin set\ p_n$ 
by (auto intro: split-list-first-propE)

with that(3,4)
obtain  $p_n'$  where p_n'-props:  $g \vdash n -p_n' \rightarrow n_z\ g \vdash m -p_m1 @ [n_z] \rightarrow n_z\ prefixeq\ p_n'\ p_n$ 
 $n_z \notin set\ (butlast\ p_n')$ 
by (meson old.path2-prefix-ex old.path2-split(1))

```

```

from V-props(3) reachable-same-var[OF that(1)] reachable-same-var[OF that(2)]
have phis-V:  $var\ g\ \varphi_r = V\ var\ g\ \varphi_s = V$  by simp-all
from reachable-props(1) that(1,2) phi-props(2) phiArg-in-allVars
have phis-allVars:  $\varphi_r \in allVars\ g\ \varphi_s \in allVars\ g$  by (metis rtranclp.cases)+

```

Various inequalities for proving paths aren't trivial.

```

have  $n \neq defNode\ g\ \varphi_r\ m \neq defNode\ g\ \varphi_r$ 
using phi-node-no-defs phis-V(1) phis-allVars(1) r-s-path-props(1,3) reachable-props(2)
that(1) by blast+

```



**from**  $\varphi$ -node-no-defs reachable-props(2) that(2) r-s-path-props(1,3) phis-V(2) that phis-allVars  
**have**  $m \neq \text{defNode } g \ \varphi_s \ n \neq \text{defNode } g \ \varphi_s$  **by** blast+

With this scenario, since  $\text{set } (\text{butlast } p_n) \cap \text{set } (\text{butlast } p_m) = \{\}$ , one of the paths  $p_n$  and  $p_m$  must end somewhere within the other, however this means the  $\phi$ -function in that node must either be  $\varphi$  or  $\varphi_r$ .

**from** *assms*  $n_z$ -props  
**consider**  $(p_n\text{-ends-in-}p_m) \ n_z = \text{defNode } g \ \varphi_s \mid (p_m\text{-ends-in-}p_n) \ n_z = \text{defNode } g \ \varphi_r$   
**proof** (cases  $n_z = \text{last } p_n$ )  
  **case** True  
    **with**  $\langle g \vdash n - p_n \rightarrow \text{defNode } g \ \varphi_r \rangle$   
    **have**  $n_z = \text{defNode } g \ \varphi_r$  **using** *old.path2-last* **by** *auto*  
    **with** *that(2)* **show** ?thesis.  
  **next**  
    **case** False  
    **from**  $n_z$ -props(2)  
    **have**  $n_z \in \text{set } p_m$  **by** *simp*  
    **with** *False*  $n_z$ -props(1)  $\langle \text{set } (\text{butlast } p_m) \cap \text{set } (\text{butlast } p_n) = \{\} \rangle \langle g \vdash m - p_m \rightarrow \text{defNode } g \ \varphi_s \rangle$   
    **have**  $n_z = \text{defNode } g \ \varphi_s$  **by** (*metis disjoint-elem not-in-butlast old.path2-last*)  
    **with** *that(1)* **show** ?thesis.  
**qed**

**thus**  $\exists \varphi_z \in \text{reachable } g \ \varphi. \exists ns \ ms. \text{old.pathsConverge } g \ n \ ns \ m \ ms \ (\text{defNode } g \ \varphi_z)$   
**proof** (cases)  
  **case**  $p_n\text{-ends-in-}p_m$   
    **have** *old.pathsConverge*  $g \ n \ p_n' \ m \ p_m \ (\text{defNode } g \ \varphi_s)$   
    **proof** (*rule old.pathsConvergeI*)  
      **from**  $p_n\text{-ends-in-}p_m \ p_n'\text{-props}(1)$  **show**  $g \vdash n - p_n' \rightarrow \text{defNode } g \ \varphi_s$  **by** *simp*  
      **from**  $\langle n \neq \text{defNode } g \ \varphi_s \rangle \ p_n\text{-ends-in-}p_m \ p_n'\text{-props}(1)$  *old.path2-nontriv* **show**  $1 < \text{length } p_n'$  **by** *auto*  
      **from** *that(4)* **show**  $g \vdash m - p_m \rightarrow \text{defNode } g \ \varphi_s$ .  
      **with**  $\langle m \neq \text{defNode } g \ \varphi_s \rangle$  *old.path2-nontriv* **show**  $1 < \text{length } p_m$  **by** *simp*  
      **from** *that*  $p_n'\text{-props}(3)$  **show**  $\text{set } (\text{butlast } p_n') \cap \text{set } (\text{butlast } p_m) = \{\}$   
      **by** (*meson butlast-prefixeq disjointI disjoint-elem in-prefix*)  
    **qed**  
    **with** *that(1,2,3)* **show** ?thesis **by** (*auto intro:reachable.intros(2)*)  
  **next**  
    **case**  $p_m\text{-ends-in-}p_n$   
    **have** *old.pathsConverge*  $g \ n \ p_n' \ m \ (p_m1 @ [n_z]) \ (\text{defNode } g \ \varphi_r)$   
    **proof** (*rule old.pathsConvergeI*)  
      **from**  $p_m\text{-ends-in-}p_n \ p_n'\text{-props}(1,2)$  **show**  $g \vdash n - p_n' \rightarrow \text{defNode } g \ \varphi_r \ g \vdash m - p_m1 @ [n_z] \rightarrow \text{defNode } g \ \varphi_r$  **by** *simp-all*  
      **with**  $\langle n \neq \text{defNode } g \ \varphi_r \rangle \langle m \neq \text{defNode } g \ \varphi_r \rangle$  **show**  $1 < \text{length } p_n' \ 1 < \text{length } (p_m1 @ [n_z])$

```

using old.path2-nontriv[of  $g\ m\ p_m1\ @\ [n_z]$ ] old.path2-nontriv[of  $g\ n$ ] by simp-all
from  $n_z$ -props  $p_n'$ -props( $\mathcal{P}$ ) show  $set\ (butlast\ p_n') \cap set\ (butlast\ (p_m1\ @\ [n_z]))$ 
= {}
using butlast-snoc disjointI in-prefix in-set-butlastD by fastforce
qed
with that(1) show ?thesis by (auto intro:reachable.intros)
qed
qed

```

Since the reachable  $g$  set was built starting at a single  $\phi$ , these paths must at some point converge *within reachable  $g$*   $\varphi$ .

```

from assign-nodes-props( $\mathcal{P}, 2$ ) ind-props V-props( $\mathcal{P}$ )  $\varphi_r$ - $V$   $\varphi_r$ -allVars
have  $\exists \varphi_z \in reachable\ g\ \varphi.$   $\exists ns\ ms.$  old.pathsConverge  $g\ n\ ns\ m\ ms$  (defNode  $g\ \varphi_z$ )
proof (induction arbitrary: p_m p_n rule: reachable.induct)
case refl

```

In the induction basis, we know that  $\varphi = \varphi_s$ , and a path to  $\varphi_r$  must be obtained – for this we need a second induction.

```

from refl.premis refl.hyps show ?case
proof (induction arbitrary: p_m p_n rule: reachable.induct)
case refl

```

The first case, in which  $\varphi_r = \varphi_s = \varphi$ , is trivial –  $\varphi$  suffices.

```

have old.pathsConverge  $g\ n\ p_n\ m\ p_m$  (defNode  $g\ \varphi$ )
proof (rule old.pathsConvergeI)
show  $1 < length\ p_n$   $1 < length\ p_m$ 
using refl V-props simpleDefs-phiDefs-var-disjoint unfolding unnecessaryPhi-def
by (metis domD domIff old.path2-hd-in- $\alpha$ n old.path2-nontriv phi-phiDefs
r-s-path-props(1) r-s-path-props(3)+)
show  $g \vdash n - p_n \rightarrow defNode\ g\ \varphi$   $g \vdash m - p_m \rightarrow defNode\ g\ \varphi$   $set\ (butlast\ p_n) \cap set$ 
 $(butlast\ p_m) = \{\}$ 
using refl by auto
qed
with  $\langle \varphi \in reachable\ g\ \varphi \rangle$  show ?case by auto
next
case (step  $\varphi'\ \varphi_r$ )

```

In this case we have that  $\varphi = \varphi_s$  and need to acquire a path going to  $\varphi_r$ , however with the aux. lemma we have, we still need that  $p_n$  and  $p_m$  are disjoint.

```

thus ?case
proof (cases set p_n \cap set p_m = \{\})
case paths-cross: False
with step reachable.intros

```

```

show ?thesis using path-crossing-yields-convergence[of  $\varphi_r \ \varphi \ p_n \ p_m$ ] by (metis
disjointI disjoint-elem)
next
case True

```

If the paths are intersection-free, we can apply our path extension lemma to obtain the path needed.

```

from step(9,8,10)  $\langle \varphi \in \text{allVars } g \rangle$  r-s-path-props(1,3) step(6,5) True step(2)
obtain ns where  $g \vdash n - p_n @ ns \rightarrow \text{defNode } g \ \varphi' \ \text{set } (\text{butlast } (p_n @ ns)) \cap \text{set } p_m$ 
= {} by (rule phiArg-disjoint-paths-extend)

from  $\langle \text{set } (\text{butlast } (p_n @ ns)) \cap \text{set } p_m = \{\} \rangle$  have  $\text{set } (\text{butlast } p_m) \cap \text{set } (\text{butlast } (p_n @ ns)) = \{\}$ 
using in-set-butlastD by fastforce
moreover
from phiArg-same-var step.hyps(2) step.prem(5) have  $\text{var } g \ \varphi' = V$ 
by auto
moreover
have  $\varphi' \in \text{allVars } g$ 
by (metis  $\varphi$ -props(2) phiArg-in-allVars reachable.cases step.hyps(1))
ultimately
show  $\exists \varphi_z \in \text{reachable } g \ \varphi. \exists ns \ ms. \text{old.pathsConverge } g \ n \ ns \ m \ ms \ (\text{defNode } g \ \varphi_z)$ 
using step.prem(1)  $\varphi$ -props V-props  $\langle g \vdash n - p_n @ ns \rightarrow \text{defNode } g \ \varphi' \rangle$ 
by  $\neg$ (rule step.IH; blast)
qed
qed
next
case (step  $\varphi' \ \varphi_s$ )

```

With the induction basis handled, we can finally move on to the induction proper.

```

show ?thesis
proof (cases  $\text{set } p_m \cap \text{set } p_n = \{\}$ )
case True
have  $\varphi_s - V: \text{var } g \ \varphi_s = V$  using step(1,2,3,9) reachable-same-var by (simp add:
phiArg-same-var)
from step(2) have  $\varphi_s$ -allVars:  $\varphi_s \in \text{allVars } g$  by (rule phiArg-in-allVars)

obtain  $p_m'$  where  $g \vdash m - p_m @ p_m' \rightarrow \text{defNode } g \ \varphi' \ \text{set } (\text{butlast } (p_m @ p_m'))$ 
 $\cap \text{set } (\text{butlast } p_n) = \{\}$ 
by (rule phiArg-disjoint-paths-extend[of  $g \ \varphi_s \ V \ \varphi_r \ m \ n \ p_m \ p_n \ \varphi'$ ])
(metis  $\varphi_s - V \ \varphi_s$ -allVars step r-s-path-props(1,3) True disjoint-iff-not-equal
in-set-butlastD)+

from step(5) this(1) step(7) this(2) step(9) step(10) step(11)

```

```

    show ?thesis by (rule step.IH[of pm@pm' pn])
  next
    case paths-cross: False
    with step.reachable.intros
    show ?thesis using path-crossing-yields-convergence[of φr φs pn pm] by blast
  qed
qed

  then obtain φz ns ms where φz ∈ reachable g φ and old.pathsConverge g n ns m
ms (defNode g φz)
  by blast
  moreover
  with reachable-props have var g φz = V by (metis V-props(3) phiArg-trancl-same-var
rtranclpD)
  ultimately have necessaryPhi' g φz using r-s-path-props
  unfolding necessaryPhi-def by blast
  moreover with ⟨φz ∈ reachable g φ⟩ have unnecessaryPhi g φz by -(rule reachable-props)
  ultimately show False unfolding unnecessaryPhi-def by blast
qed
qed

```

Finally, to conclude, we'll show that the above theorem is indeed a stronger assertion about a graph than the lack of trivial  $\phi$ -functions. Intuitively, this is because a set containing only a trivial  $\phi$ -function is a redundant set.

```

corollary
assumes ¬(∃ P. redundant-set g P)
shows ¬redundant g
proof -
  have redundant g ⇒ ∃ P. redundant-set g P
  proof -
    assume redundant g
    then obtain φ where phi g φ ≠ None trivial g φ
    unfolding redundant-def redundant-set-def dom-def phiArg-def trivial-def isTrivialPhi-def
    by (clarsimp split: option.splits) fastforce
    hence redundant-set g {φ}
    unfolding redundant-set-def dom-def phiArg-def trivial-def isTrivialPhi-def
    by auto
    thus ?thesis by auto
  qed
  with assms show ?thesis by auto
qed

```

end

end

## A.2. Implementation of the Peeling Algorithm

```

1  /*
2  * This file is part of libFirm.
3  * Copyright (C) 2016 Karlsruhe Institute of Technology
4  */
5
6  /**
7  * @file
8  * @brief Unnecessary Phi SCC removal.
9  * @date 13.03.2016
10 * @author Max Wagner
11 * @brief
12 * Removal of Phi SCCs which have at most one true predecessor.
13 * See "Simple and Efficient Construction of Static Single Assignment
14 * Form" by Braun et al.
15 */
16 #include <irdump_t.h>
17 #include <bits/time.h>
18 #include <time.h>
19 #include "debug.h"
20 #include "ircons.h"
21 #include "irgmod.h"
22 #include "irgwalk.h"
23 #include "irnodelist.h"
24 #include "irnodeset.h"
25 #include "irtools.h"
26 #include "util.h"
27
28
29 /** We use (yet another implementation of) Tarjan's algorithm to find
30 * SCCs, which implicitly obtains them
31 * in reverse topological order. (which forgoes the need for a fixpoint
32 * iteration)
33 * These SCCs are then checked for whether they are, as a whole,
34 * redundant. If they are, we mark the mapping
35 * from nodes in the SCC to their unique non-SCC predecessor for edge
36 * rerouting later.
37 *
38 * If an SCC is not redundant, we still have to check all SCCs in the
39 * subgraph induced by the SCC (removing any nodes that
40 * connect to its outside from the working set). In order to do this,
41 * we note the "scc id" of each node
42 * and only increase this number for the nodes we may recurse on. (
43 * since the "inner" part of different SCCs are
44 * disconnected, this works out on the whole)
45 *
46 * SCCs are stored in a doubly-linked list, with each SCC consisting of
47 * an ir_nodeset of nodes.
48 */

```

```
42 typedef struct scc {
43     list_head    link;
44     ir_nodese_t nodes;
45     unsigned    depth;
46 } scc_t;
47
48 typedef struct scc_env {
49     struct obstack    obst;
50     ir_node           **stack;
51     size_t           stack_top;
52     unsigned         next_index;
53     list_head        working_set_sccs; /**< the sccs we *just* found,
54     and haven't yet evaluated */
55     list_head        scc_work_stack; /**< the sets of nodes we still
56     need to evaluate in future iterations */
57     ir_nodehashmap_t replacement_map; /**< map from node to their
58     replacement */
59 } scc_env_t;
60
61 typedef struct scc_irn_info {
62     bool            in_stack; /**< Marks whether node is on the stack
63     . */
64     unsigned        dfn; /**< Depth first search number. */
65     unsigned        uplink; /**< dfn number of ancestor. */
66     unsigned        depth; /**< iteration depth of scc search */
67 } scc_irn_info_t;
68
69 static scc_irn_info_t *get_irn_info(ir_node *node, scc_env_t *env)
70 {
71     scc_irn_info_t *e = get_irn_link(node);
72     if (e == NULL) {
73         e = OALLOCZ(&env->obst, scc_irn_info_t);
74         node->link = e;
75     }
76     return e;
77 }
78
79 /**
80 * push a node onto the stack, potentially growing it
81 *
82 * @param env the algorithm environment
83 * @param node the node to push
84 */
85 static void push(scc_env_t *env, ir_node *node)
86 {
87     if (env->stack_top == ARR_LEN(env->stack)) {
88         size_t nlen = ARR_LEN(env->stack) * 2;
89         ARR_RESIZE(ir_node*, env->stack, nlen);
90     }
91     env->stack[env->stack_top++] = node;
92     scc_irn_info_t *e = get_irn_info(node, env);
```

```

90     e->in_stack = true;
91 }
92 }
93
94 /**
95  * pop a node from the stack
96  *
97  * @param env  the algorithm environment
98  * @return The topmost node
99  */
100 static ir_node *pop(scc_env_t *env)
101 {
102     ir_node      *n = env->stack[--env->stack_top];
103     scc_irn_info_t *e = get_irn_info(n, env);
104     e->in_stack = false;
105     return n;
106 }
107
108
109 /** return the unique predecessor of a redundant scc, or NULL if the
110     scc is not redundant.
111  * (Also marks nodes elidable for next iteration by clearing their dfn
112     and setting their depth)
113  */
114 static ir_node *get_unique_pred(scc_t *scc, scc_env_t *env) {
115     ir_node *unique_pred = NULL;
116     bool redundant = true;
117     foreach_irn_in(scc->nodes, irn, iter) {
118         // only nodes which are not on the "rim" of the scc are
119         // eligible for the next iteration
120         bool eligible_for_next_iteration = true;
121         foreach_irn_in(scc->nodes, idx, original_pred) {
122             // we can safely ignore self-loops in this regard
123             if (original_pred != irn) {
124                 // previous iterations might have "deleted" the node
125                 // already.
126                 ir_node *pred = ir_nodehashmap_get(ir_node, &env->
127                     replacement_map, original_pred);
128                 if (pred == NULL) pred = original_pred;
129
130                 if (!ir_nodehashmap_contains(&scc->nodes, pred)) {
131                     if (unique_pred && unique_pred != pred) redundant =
132                         false;
133                     // we don't break out of the loop because we still
134                     // want to mark all necessary nodes
135                     unique_pred = pred;
136                     eligible_for_next_iteration = false;
137                 }
138             }
139         }
140     }
141     if (eligible_for_next_iteration) {

```

```

135         scc_irn_info_t *info = get_irn_info(irn, env);
136         info->depth++;
137         scc->depth = info->depth;
138         info->dfn = 0;
139     }
140 }
141 return redundant ? unique_pred : NULL;
142 }
143
144
145 /** Append the working set to the work queue and prime the first
146  * eligible SCC in the work queue for the next iteration
147  */
148 static void prepare_next_iteration(scc_env_t *env) {
149
150     list_splice_init(&env->working_set_sccs, &env->scc_work_stack);
151
152     list_for_each_entry_safe(scc_t, scc, tmp, &env->scc_work_stack,
153                             link) {
154         ir_node *unique_pred = get_unique_pred(scc, env);
155         if (unique_pred) {
156             // SCC is redundant, reroute and discard
157             foreach_ir_nodeset(&scc->nodes, irn, iter) {
158                 ir_nodehashmap_insert(&env->replacement_map, irn,
159                                       unique_pred);
160             }
161             ir_nodeset_destroy(&scc->nodes);
162             list_del_init(&scc->link);
163         } else {
164             foreach_ir_nodeset(&scc->nodes, irn, iter) {
165                 // get_unique_pred has marked all "inner" nodes by
166                 // resetting their dfn, the rest must be removed.
167                 if (get_irn_info(irn, env)->dfn != 0)
168                     ir_nodeset_remove_iterator(&scc->nodes, &iter);
169             }
170
171             if (ir_nodeset_size(&scc->nodes) > 1) break;
172             else {
173                 // we have no need for this scc anymore
174                 ir_nodeset_destroy(&scc->nodes);
175                 list_del_init(&scc->link);
176             }
177         }
178     }
179 }
180
181 static inline bool is_removable(ir_node *irn, scc_env_t *env, unsigned
182 depth) {
183     scc_irn_info_t *info = get_irn_info(irn, env);
184     return is_Phi(irn) && !get_Phi_loop(irn) && info->depth >= depth;
185 }

```



```

182
183 /** Perform's Tarjan's algorithm, starting at a given node
184 *
185 * returns false if n must be ignored
186 * (either because it's not a Phi node or because it's been excluded
187   in a previous run) */
187 static bool find_scc_at(ir_node *n, scc_env_t *env, unsigned depth)
188 {
189     if (!is_removable(n, env, depth)) return false;
190
191     scc_irn_info_t *info = get_irn_info(n, env);
192     if (info->dfn != 0) {
193         // node has already been visited
194         return true;
195     }
196     info->dfn = ++env->next_index;
197     info->uplink = info->dfn;
198     push(env, n);
199     info->in_stack = true;
200     foreach_irn_in(n, i, pred) {
201         // the node might have been identified as part of a redundant
202         scc already, so we need to check
202         ir_node *canonical_pred = ir_nodehashmap_get(ir_node, &env->
203             replacement_map, pred);
203         if (!canonical_pred) canonical_pred = pred;
204
205         scc_irn_info_t *pred_info = get_irn_info(canonical_pred, env);
206         if (pred_info->dfn == 0 && find_scc_at(canonical_pred, env,
207             depth)) {
207             info->uplink = MIN(pred_info->uplink, info->uplink);
208         } else if (pred_info->in_stack) {
209             info->uplink = MIN(pred_info->dfn, info->uplink);
210         }
211     }
212     if (info->dfn == info->uplink) {
213         // found an scc
214         struct scc *scc = OALLOC(&env->obst, struct scc);
215         ir_nodelist_init(&scc->nodes);
216
217         ir_node *n2;
218         do {
219             n2 = pop(env);
220             scc_irn_info_t *n2_info = get_irn_info(n2, env);
221             n2_info->in_stack = false;
222             ir_nodelist_insert(&scc->nodes, n2);
223             scc->depth = n2_info->depth;
224         } while (n2 != n);
225         list_add_tail(&scc->link, &env->working_set_sccs);
226     }
227     return true;
228 }
229

```

```
230 // One recursive "find_scc_at" handles a complete phi web, but there
      // may be many, so we need to walk the graph
231 static void _start_walk(ir_node *irn, void *env) {
232     find_scc_at(irn, (scc_env_t *) env, 0 /* this is only used for the
      // initial SCC search, so depth 0 is fine*/);
233 }
234
235 FIRM_API void opt_remove_unnecessary_phi_sccs(ir_graph *irg)
236 {
237     struct scc_env env;
238     memset(&env, 0, sizeof(env));
239     struct obstack temp;
240     obstack_init(&temp);
241     env.obst = temp;
242     env.stack = NEW_ARR_F(ir_node*, 128);
243     ir_nodehashmap_init(&env.replacement_map);
244     INIT_LIST_HEAD(&env.working_set_sccs);
245     INIT_LIST_HEAD(&env.scc_work_stack);
246
247     ir_reserve_resources(irg, IR_RESOURCE_IRN_LINK);
248     irg_walk_graph(irg, NULL, firm_clear_link, NULL);
249
250     // populate work queue with an initial round of SCCs
251     irg_walk_graph(irg, _start_walk, NULL, &env);
252     prepare_next_iteration(&env);
253
254     while (!list_empty(&env.scc_work_stack)) {
255         // pop an SCC from the front of the queue and evaluate it
256         scc_t *current_set = list_entry(env.scc_work_stack.next, scc_t,
      // link);
257         list_del(env.scc_work_stack.next);
258         foreach_ir_node_set(&current_set->nodes, irn, iter) {
259             find_scc_at(irn, &env, current_set->depth);
260         }
261         // clean up the scc we just popped off
262         ir_node_set_destroy(&current_set->nodes);
263         prepare_next_iteration(&env);
264     }
265
266     ir_nodehashmap_entry_t entry;
267     ir_nodehashmap_iterator_t iter;
268
269     DEBUG_ONLY (if (ir_nodehashmap_size(&env.replacement_map))
      // dump_ir_graph(irg, "PRE"););
270
271     foreach_ir_nodehashmap(&env.replacement_map, entry, iter) {
272         exchange(entry.node, (ir_node *) entry.data);
273     }
274
275     DEBUG_ONLY (if (ir_nodehashmap_size(&env.replacement_map))
      // dump_ir_graph(irg, "POST"););
276
```

```

277
278     ir_nodemap_destroy((ir_nodemap_t *) &env.replacement_map);
279     DEL_ARR_F(env.stack);
280     obstack_free(&env.obst, NULL);
281     ir_free_resources(irg, IR_RESOURCE_IRN_LINK);
282
283 }

```

## A.3. Code Used for Worst-Case Example

Note that libFIRM automatically optimizes IR nodes as they're created. To prevent libFIRM from optimizing the (in this case obvious) redundant  $\phi$ -functions on creation, we also had to temporarily disable this optimization for  $\phi$  nodes. A more elaborate setup for creating test graphs could have avoided this though.

```

1
2  ir_graph *create_blank_graph(void) {
3      ir_type *t = new_type_method(0, 1, false, 0, mtp_no_property);
4      set_method_res_type(t, 0, new_type_primitive(get_models()));
5      ir_entity *ent = new_entity(get_glob_type(), new_id_from_str("test_
        "), t);
6      ir_graph *g = new_ir_graph(ent, 100);
7      return g;
8  }
9
10 ir_graph *create_ladder_graph(int steps) {
11     ir_graph *g = create_blank_graph();
12     set_current_ir_graph(g);
13
14     ir_node *startbl = get_irg_start_block(g);
15     set_cur_block(startbl);
16
17     ir_node *n0 = new_Const_long(mode_ls, 0);
18     ir_node *n1 = new_Const_long(mode_ls, 1);
19
20     ir_node *ins1[] = {n0};
21     ir_node *ret = new_r_Return(startbl, get_irg_initial_mem(g), 1,
        ins1);
22     add_immBlock_pred(get_irg_end_block(g), ret);
23
24     ir_node *ins[] = {n0, n1};
25     ir_node *final0 = new_r_Phi(startbl, 2, ins, mode_ls);
26     ir_node *final1 = new_r_Phi(startbl, 2, ins, mode_ls);
27
28     ins[0] = final0;
29     ins[1] = final1;
30
31     ir_node *final = new_r_Phi(startbl, 2, ins, mode_ls);
32
33     ir_node *phi0, *phi1;

```

```
34     ir_node *oldphi0 = n0, *oldphi1 = n1;
35     // Loop creating n steps
36     for (int n = 0; n < steps; n++) {
37         ins[0] = oldphi0;
38         ins[1] = final0;
39         phi0 = new_r_Phi(startbl, 2, ins, mode_ls);
40         ins[0] = oldphi1;
41         ins[1] = final1;
42         phi1 = new_r_Phi(startbl, 2, ins, mode_ls);
43
44         oldphi0 = phi0;
45         oldphi1 = phi1;
46     }
47
48     ir_node *fixup[] = {phi0, final1};
49     set_irn_in(final0, 2, fixup);
50     fixup[0] = phi1;
51     fixup[1] = final0;
52     set_irn_in(final1, 2, fixup);
53
54     set_irn_n(ret, 1, final);
55     clear_irg_constraints(g, IR_GRAPH_CONSTRAINT_CONSTRUCTION);
56     return g;
57 }
```